Programming Microsoft Windows with C#
by Charles Petzold

Microsoft Press © 2002

For the great people from [OR] Forum

Enjoy
Programming Microsoft Windows with C#

Charles Petzold

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2002 by Charles Petzold

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Petzold, Charles, 1953-
Programming Microsoft Windows with C# / Charles Petzold.
p. cm.
Includes index.

QA76.73.C154 P48 2001
005.26’8--dc21

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress.

Send comments to mspinput@microsoft.com.

ClearType, Georgia, IntelliMouse, Microsoft, Microsoft Press, MS-DOS, MSDN, Natural, the .NET logo, OpenType, Verdana, Visual Basic, Visual C#, Visual Studio, Webdings, Win32, Windows, Windows NT, and Wingdings are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: Danielle Bird
Project Editor: Sally Stickney
Technical Editor: Jean Ross

Body Part No. X08-22413

About The Author

Charles Petzold

Charles Petzold (www.charlespetzold.com) is a full-time freelance writer who has been programming for Microsoft Windows since 1985 and writing about Windows programming for nearly as long. He wrote the very first magazine article about Windows programming for the December 1986 issue of Microsoft Systems Journal. His book Programming Windows (first published by Microsoft Press in 1988 and currently in its fifth edition) taught a generation of programmers how to write applications
for Windows. In May 1994, Petzold was one of only seven people (and the only writer) to be given a Windows Pioneer Award from Windows Magazine and Microsoft Corporation for his contribution to the success of Microsoft Windows. He is also the author of a unique introduction to the inner workings of computers entitled Code: The Hidden Language of Computer Hardware and Software. Petzold is currently researching a book on the historical origins of software.

The manuscript for this book was prepared and galleyed using Microsoft Word version 2000. Pages were composed by Microsoft Press using Adobe PageMaker 6.52 for Windows, with text in Garamond and display type in Helvetica Condensed. Composed pages were delivered to the printer as electronic prepress files.

Cover Designer: Methodologie, Inc.

Interior Graphic Designer: James D. Kramer

Principal Compositor: Paula Gorelick

Interior Artist: Joel Panchot

Principal Copy Editor: Holly M. Viola

Indexer: Shane-Armstrong Information Systems
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>149</td>
</tr>
<tr>
<td>How to Get a Graphics Object</td>
<td>149</td>
</tr>
<tr>
<td>Pens, Briefly</td>
<td>150</td>
</tr>
<tr>
<td>Straight Lines</td>
<td>151</td>
</tr>
<tr>
<td>An Introduction to Printing</td>
<td>152</td>
</tr>
<tr>
<td>Properties and State</td>
<td>157</td>
</tr>
<tr>
<td>Anti-Aliasing</td>
<td>158</td>
</tr>
<tr>
<td>Multiple Connected Lines</td>
<td>160</td>
</tr>
<tr>
<td>Curves and Parametric Equations</td>
<td>164</td>
</tr>
<tr>
<td>The Ubiquitous Rectangle</td>
<td>167</td>
</tr>
<tr>
<td>Generalized Polygons</td>
<td>169</td>
</tr>
<tr>
<td>Easier Ellipses</td>
<td>170</td>
</tr>
<tr>
<td>Arcs and Pies</td>
<td>171</td>
</tr>
<tr>
<td>Filling Rectangles, Ellipses, and Pies</td>
<td>177</td>
</tr>
<tr>
<td>Off by 1</td>
<td>178</td>
</tr>
<tr>
<td>Polygons and the Filling Mode</td>
<td>180</td>
</tr>
</tbody>
</table>

**CHAPTER 6: TAPPING INTO THE KEYBOARD** ........................................ 184

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>184</td>
</tr>
<tr>
<td>Ignoring the Keyboard</td>
<td>184</td>
</tr>
<tr>
<td>Who's Got the Focus?</td>
<td>184</td>
</tr>
<tr>
<td>Keys and Characters</td>
<td>185</td>
</tr>
<tr>
<td>Keys Down and Keys Up</td>
<td>186</td>
</tr>
<tr>
<td>The Keys Enumeration</td>
<td>187</td>
</tr>
<tr>
<td>Testing the Modifier Keys</td>
<td>194</td>
</tr>
<tr>
<td>Reality Check</td>
<td>195</td>
</tr>
<tr>
<td>A Keyboard Interface for SYSINFO</td>
<td>196</td>
</tr>
<tr>
<td>Keypress for Characters</td>
<td>197</td>
</tr>
<tr>
<td>Control Characters</td>
<td>198</td>
</tr>
<tr>
<td>Looking at the Keys</td>
<td>199</td>
</tr>
<tr>
<td>Invoking the Win32 API</td>
<td>203</td>
</tr>
<tr>
<td>Handling Input from Foreign Keyboards</td>
<td>205</td>
</tr>
<tr>
<td>Input Focus</td>
<td>207</td>
</tr>
<tr>
<td>The Missing Caret</td>
<td>208</td>
</tr>
<tr>
<td>Echoing Key Characters</td>
<td>212</td>
</tr>
<tr>
<td>Right-to-Left Problems</td>
<td>216</td>
</tr>
</tbody>
</table>

**CHAPTER 7: PAGES AND TRANSFORMS** ............................................. 218

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>218</td>
</tr>
<tr>
<td>Device Independence Through Text</td>
<td>218</td>
</tr>
<tr>
<td>How Much Is That in Real Money?</td>
<td>218</td>
</tr>
<tr>
<td>Dots Per Inch</td>
<td>220</td>
</tr>
<tr>
<td>What's with the Printer?</td>
<td>221</td>
</tr>
<tr>
<td>Manual Conversions</td>
<td>222</td>
</tr>
<tr>
<td>Page Units and Page Scale</td>
<td>225</td>
</tr>
<tr>
<td>Pen Widths</td>
<td>228</td>
</tr>
<tr>
<td>Page Transforms</td>
<td>231</td>
</tr>
</tbody>
</table>
CHAPTER 8: TAMING THE MOUSE ................................................................. 254

OVERVIEW ........................................................................................................ 254
THE DARK SIDE OF THE MOUSE ................................................................. 254
IGNORING THE MOUSE ..................................................................................... 255
SOME QUICK DEFINITIONS ............................................................................. 255
INFORMATION ABOUT THE MOUSE ........................................................... 255
THE MOUSE WHEEL .......................................................................................... 256
THE FOUR BASIC MOUSE EVENTS ................................................................. 257
DOING THE WHEEL ........................................................................................... 259
MOUSE MOVEMENT .......................................................................................... 262
TRACKING AND CAPTURING THE MOUSE ............................................... 264
ADVENTURES IN TRACKING ............................................................................ 267
GENERALIZING CODE WITH INTERFACES ................................................. 274
CLICKS AND DOUBLE-ClickS ......................................................................... 277
MOUSE-RELATED PROPERTIES ....................................................................... 278
ENTERING, LEAVING, HOVERING ................................................................. 279
THE MOUSE CURSOR ......................................................................................... 280
AN EXERCISE IN Hit-TESTING ......................................................................... 287
ADDING A KEYBOARD INTERFACE ............................................................... 289
PUTTING THE CHILDREN TO WORK .............................................................. 291
Hit-TESTING TEXT ............................................................................................ 296
SCRIBBLING WITH THE MOUSE ..................................................................... 297

CHAPTER 9: TEXT AND FONTS ..................................................................... 303

OVERVIEW ........................................................................................................ 303
FOhNTS UNDER WINDOWS ............................................................................... 303
TALKING TYPE .................................................................................................. 303
FONT HEIGHTS AND LINE SPACING ............................................................... 305
DEFAULT FONTS ............................................................................................... 306
VARIATION ON A FONT ...................................................................................... 306
CREATING FONTS BY NAME ........................................................................... 309
A POINT SIZE BY ANY OTHER NAME ............................................................. 313
CLASH OF THE UNITS ....................................................................................... 317
FONT PROPERTIES AND METHODS ............................................................... 319
NEW FONTS FROM FontFAMILY ....................................................................... 324
Introduction

This book shows you how to write programs that run under Microsoft Windows. There are a number of ways to write such programs. In this book, I use the new object-oriented programming language C# (pronounced "C sharp") and a modern class library called Windows Forms. The Windows Forms class library is part of the Microsoft .NET ("dot net") Framework unveiled in the summer of 2000 and introduced about a year and a half later.

The Microsoft .NET Framework is an extensive collection of classes that provides programmers with much of what they need to write Internet, Web, and Windows applications. Much of the media coverage of .NET has focused on the Web programming. This book discusses the other part of .NET. You use Windows Forms to write traditional stand-alone Windows applications (what are now sometimes called client applications) or front ends for distributed applications.

Windows Forms provides almost everything you need to write full-fledged Windows applications. The big omission is multimedia support. There’s not even a Windows Forms function to beep the computer’s speaker! I was tempted to write my own multimedia classes but restrained myself under the assumption (reasonable, I hope) that the next release of Windows Forms will include multimedia support that is flexible, powerful, and easy to use.

The classes defined in the .NET Framework are language-neutral. Microsoft has released new versions of C++ and Visual Basic that can use these classes, as well as the new programming language C#. Other language vendors are adapting their own languages to use the .NET classes. These new compilers (either optionally or by default) convert source code to an intermediate language in an .exe file. At runtime, the intermediate language is compiled into appropriate microprocessor machine code. Thus, the .NET Framework is potentially platform independent.

I chose to use C# for this book because C# and .NET were—in a very real sense—made for each other. Because of the language-neutral aspect of the .NET Framework, you may be able to use this book to learn how to write Windows Forms applications with other .NET languages.

Windows Programming: An Overview

Microsoft released the first version of Windows in the fall of 1985. Since then, Windows has been progressively updated and enhanced, most dramatically in Windows NT (1993) and Windows 95 (1995), when Windows moved from a 16-bit to a 32-bit architecture.

When Windows was first released, there was really only one way to write Windows applications, and that was by using the C programming language to access the Windows application programming interface (API). Although it was also possible to access the Windows API using Microsoft Pascal, this approach was rarely used.

Over the years, many other languages have been adapted for doing Windows programming, including Visual Basic and C++. Both C++ and C# are object-oriented languages that support most of the types, operators, expressions, and statements of the C programming language. For this reason, C++ and C# (as well as Java) are sometimes called C-based languages, or languages of the C family.

With the introduction of .NET, Microsoft currently offers three approaches to writing Windows applications using a C-based language:

<table>
<thead>
<tr>
<th>Year Introduced</th>
<th>Language</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>C</td>
<td>Windows application programming interface (API)</td>
</tr>
<tr>
<td>1992</td>
<td>C++</td>
<td>Microsoft Foundation Class (MFC) Library</td>
</tr>
<tr>
<td>2001</td>
<td>C# or C++</td>
<td>Windows Forms (part of the .NET Framework)</td>
</tr>
</tbody>
</table>
It's not my job to tell you what language or interface you should use to write Windows applications. That's a decision only you can make based on the particular programming job and the resources available to you.

If you want to learn more about the Windows API, many people have found my book *Programming Windows* (5th edition, Microsoft Press, 1998) to be valuable.

I never liked MFC. Ever since it was introduced, I thought it was poorly designed and barely object oriented. Consequently, I've never written about MFC. But that's only a personal view. Many other programmers have successfully used MFC, and it's currently one of the most popular approaches to Windows programming. A good place to learn about MFC is the book *Programming Windows with MFC* (2nd edition, Microsoft Press, 1999) by Jeff Prosise. For the more advanced Windows programmer, I also want to recommend *Programming Applications for Microsoft Windows* (Microsoft Press, 1999) by Jeffrey Richter.

From my view, Windows Forms is much better designed than MFC and much closer to what I envision as an ideal object-oriented interface to Windows. Over the past 14 months that I've been working on this book, it has become my preferred approach to Windows programming.

Programmatically speaking, both the MFC and Windows Forms interfaces work by making calls to the Windows API. Architecturally, they can be said to sit on top of the API. These higher-level interfaces are intended to make Windows programming easier. Generally, you can do specific tasks in MFC or Windows Forms with fewer statements than when using the API.

While high-level interfaces such as MFC or Windows Forms often improve the programmer's productivity, any interface that makes use of another interface is obviously less versatile than the underlying interface. You can do many things using the Windows API that you can't do using the Windows Forms classes.

Fortunately, with a little extra work, you can make calls to the Windows API from a Windows Forms program. Only occasionally in this book did I come across an omission in the .NET Framework so profound that I needed to make use of this facility. My overall philosophy has been to respect the insulation that Windows Forms offers from the inner workings of Windows itself.

**User Requirements**

To use this book most profitably, you need to be able to compile and run C# programs. To compile the programs, you need a C# compiler. To run these programs, you need the .NET runtime (called the common language runtime, or CLR), which is a collection of dynamic-link libraries.

Both these items are included in Microsoft Visual C#, a modern integrated development environment. Alternatively, you can purchase the more extensive and more expensive Microsoft Visual Studio .NET, which will also let you program in C++ and Visual Basic in addition to C#.

If you prefer a more rugged approach, you can instead download the free .NET Framework software development kit (SDK). The download includes a command-line C# compiler and the .NET runtime. First go to [http://msdn.microsoft.com/downloads](http://msdn.microsoft.com/downloads). At the left, select Software Development Kits, and then look for the .NET Framework. (Keep in mind that this Web site, as with all the Web sites mentioned throughout this book, could change, move, or in some cases disappear completely, at any time.)

I've written this book under the assumption that you at least know how to program in C. Being familiar with C++ or Java is helpful but not necessary. Because C# is a new language, the first chapter of this book provides a whirlwind introduction to C# and essential concepts of object-oriented programming. Throughout the rest of the book, I often take time to discuss miscellaneous C# concepts as they are encountered.

But this book doesn't provide a comprehensive tutorial for C#. If you want more background and skill in working with the language, other books on C# are available, and many others will undoubtedly become available as the language becomes more popular. The book *Inside C#* (Microsoft Press, 2001) by Tom Archer provides information on writing C# code and also on what's going on beneath the surface. *Microsoft Visual C# Step by Step* (Microsoft Press, 2001) by John Sharp and Jon Jager takes a more tutorial approach.
I sometimes make reference to the Windows API in this book. Like I said previously, you can consult my book *Programming Windows* to learn more about the API.

**System Requirements**

As I mentioned in the preceding section, to use this book effectively, you need to be able to compile and run C# programs. System requirements are as follows:

- Microsoft .NET Framework SDK (minimum); Microsoft Visual C# or Microsoft Visual Studio .NET (preferred)
- Microsoft Windows NT 4.0, Windows 2000, or Windows XP.

To run your C# programs on other computers requires that the .NET runtime (also referred to as the .NET Framework redistributable package) be installed on those machines. That package comes with the .NET Framework SDK, Visual C#, and Visual Studio .NET. The redistributable package can be installed on the versions of Windows already mentioned as well as Windows 98 and Windows Millennium Edition (Me).

If you want to install the sample files from the companion CD to your hard drive, you'll need approximately 2.1 MB of additional hard disk space. (Fully compiled, the samples use just over 20 MB.)

**The Organization of This Book**

When Windows 1.0 was first released, the entire API was implemented in three dynamic link libraries named KERNEL, USER, and GDI. Although the DLLs associated with Windows have become much more voluminous, it is still useful to divide Windows function calls (or framework classes) into these three categories: The kernel calls are those implemented in the architectural interior of the operating system, and are generally concerned with tasking, memory management, and file I/O. The term user refers to the user interface. These are functions to create windows, use menus and dialog boxes, and display controls such as buttons and scroll bars. GDI is the Graphics Device Interface, that part of Windows responsible for displaying graphical output (including text) on the screen and printer.

This book begins with four introductory chapters. Starting with Chapter 5 (which shows you how to draw lines and curves) and continuing through Chapter 24 (on the Windows clipboard), the chapters alternate between graphics topics (odd-numbered chapters) and user interface topics (even-numbered chapters).

Normally a book like this wouldn't spend much time with non-Windows topics such as file I/O, floating-point mathematics, and string manipulation. However, because the .NET Framework and C# are so new, I found myself wishing I had a coherent guide through those classes. So I wrote such guides myself. These are included as three appendices on files, math, and strings. You can consult these appendices any time after reading Chapter 1.

I've tried to order the chapters—and the topics within the chapters—so that each topic builds on succeeding topics with a minimal number of "forward references." I've written the book so that you can read it straight through, much like you'd read the uncut version of *The Stand* or *The Decline and Fall of the Roman Empire*.

Of course, it's good if a book as long as this one serves as a reference as well as a narrative. For that reason, many of the important methods, properties, and enumerations used in Windows Forms programming are listed in tables in the chapters in which they are discussed. A book of even this size cannot hope to cover everything in Windows Forms, however. It is no substitute for the official class documentation.

Windows Forms programs require little overhead, so this book includes plenty of code examples in the form of complete programs. You are free to cut and paste pieces of code from these programs into your own programs. (That's what these programs are for.) But don't distribute the code or programs as is. That's what this book is for.

The C# compiler has a terrific feature that lets you write comments with XML tags. However, I've chosen not to make use of this feature. The programs in this book tend to have few comments anyway because the code is described in the text that surrounds the programs.

As you may know, Visual C# allows you to interactively design the appearance of your applications. You position various controls (buttons, scroll bars, and so forth) on the surface of your window, and
Visual C# generates the code. While such techniques are very useful for quickly designing dialog boxes and front-panel types of applications, I have ignored that feature of Visual C# in this book.

In this book, we're not going to let Visual C# generate code for us. In this book, we're going to learn how to write our own code.

The CD-ROM

The companion CD-ROM contains all the sample programs in this book. You can load the solution files (.sln) or project files (.csproj) into Visual C# and recompile the programs.

Frankly, I've never had much use for CD-ROMs in books. When learning a new language, I prefer to type in the source code myself—even if it's someone else's source code at first. I find I learn the language faster that way. But that's just me.

If the CD-ROM is missing or damaged, don't send e-mail to me asking for a replacement. Contractually, I can't send you a new one. Microsoft Press is the sole distributor of this book and the CD-ROM. To get a replacement CD-ROM or other support information regarding this book, contact Microsoft Press. (See the "Support" section that follows for contact information.)

Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD-ROM. Microsoft Press provides corrections for books through the World Wide Web at the following address:

http://www.microsoft.com/mspress/support/

To connect directly to the Microsoft Press Knowledge Base and enter a query regarding a question or issue that you may have, go to:

http://www.microsoft.com/mspress/support/search.asp

If you have comments, questions, or ideas regarding this book or the companion CD-ROM, please send them to Microsoft Press using either of the following methods:

Postal Mail:
Microsoft Press
Attn: Programming Microsoft Windows with C# Editor
One Microsoft Way
Redmond, WA 98052-6399

E-mail:

MSPINPUT@MICROSOFT.COM

Please note that product support is not offered through the above mail addresses. For support information regarding C#, Visual Studio, or the .NET Framework, please visit the Microsoft Product Support Web site at

http://support.microsoft.com

Special Thanks

Writing is usually a very solitary job, but fortunately there are always several people who make the work much easier.

I want to thank my agent Claudette Moore of the Moore Literary Agency for getting this project going and handling all the messy legal stuff.

As usual, the folks at Microsoft Press were an absolute pleasure to work with and once again have helped prevent me from embarrassing myself. If it were not for my project editor, Sally Stickney, and my technical editor, Jean Ross, this book would be infested with gibberish and buggy code. While editors may seem superhuman at times, they are regretfully not. Any bugs or incomprehensible sentences that remain in the book are my fault and no one else’s.
Let me not forget to cite Johannes Brahms for providing musical accompaniment while I worked, and Anthony Trollope for escapist literature in the evenings.

My Sunday, Tuesday, and Thursday gatherings of friends continue to help and support me in ways that are sometimes obvious, sometimes subtle, but always invaluable.

And most of all, I want to thank my fiancée, Deirdre, for providing a very different (non .NET) framework for me in which to live, work, and love.

Charles Petzold
New York City
November, 2001
Chapter 1: Console Thyself

Overview

In that succinct and (perhaps consequently) much-beloved classic tutorial The C Programming Language, authors Brian Kernighan and Dennis Ritchie begin by presenting what has come to be known as the hello-world program:[1]

```c
#include <stdio.h>

main()
{
    printf("hello, world\n");//
}
```

While such a program hardly exploits the power of today's computers, it's certainly useful on the practical level because it gives the eager student programmer the opportunity to make sure that the compiler and all its associated files are correctly installed. The hello-world program also reveals the overhead necessary in a particular programming language: hello-world programs can be a single line in some languages but quite scary in others. The hello-world program is also helpful to the author of a programming book because it provides an initial focal point to begin the tutorial.

As all C programmers know, the entry point to a C program is a function named `main`, the `printf` function displays formatted text, and `stdio.h` is a header file that includes definitions of `printf` and other standard C library functions. The angle brackets, parentheses, and curly braces are used to enclose information or to group collections of language statements.

The traditional hello-world program is designed to be run in a programming environment that supports a quaint and old-fashioned type of text-only computer interface known as a command line, or console. This type of interface originated on a machine called the teletypewriter, which was itself based on an early word processing device known as the typewriter. As a user types on the teletypewriter keyboard, the device prints the characters on a roll of paper and sends them to a remote computer. The computer responds with characters of its own, which the teletypewriter receives and also displays on the paper. In this input/output model, there's no concept of positioning text on the page. That's why the `printf` function simply displays the text wherever the teletypewriter print head (or the cursor of a video-based command line) happens to be at the time.

A command-line interface exists in Microsoft Windows in the form of an application window called MS-DOS Prompt or Command Prompt. While the command-line interface has been largely obsoleted by graphical interfaces, command-line programs are often simpler than programs written for graphical environments, so they remain a good place to begin learning a new programming language.


The C# Version

In this book, I'll be using a programming language called C# (as in C-sharp, like the key of Beethoven's Moonlight Sonata). Designed by Anders Hejlsberg at Microsoft, C# is a modern object-oriented programming language that incorporates elements from C, C++, Java, Pascal, and even BASIC. This chapter presents a whirlwind (but necessarily incomplete) tour of the language.

C# source code files have the filename extension .cs ("c sharp"). My first C# version of the hello-world program is the file ConsoleHelloWorld.cs.

```c
//------------------------------------------------
// ConsoleHelloWorld.cs © 2001 by Charles Petzold
//------------------------------------------------
ConsoleHelloWorld.cs
```
class ConsoleHelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
    }
}

You have a couple options in compiling this program, depending on how much money you want to spend and how much modern programming convenience you wouldn't mind foregoing.

The cheapest approach is to download the .NET Framework Software Development Kit (SDK) from http://msdn.microsoft.com. Installing the SDK also installs the dynamic-link libraries (DLLs) that comprise the .NET runtime environment. The .NET technical documentation is available in a Windows-based program. You also get a command-line C# compiler that you can use to compile the programs shown in these pages.

You can use any text editor—from Microsoft Notepad on up—to write C# programs. The C# compiler is named csc.exe. You compile ConsoleHelloWorld.cs on a command line like so:
csc consolehelloworld.cs

That's it. There's no link step involved. (As you'll see in the next chapter, compiling a Windows Forms program rather than a console program requires some additional compiler arguments.) The compiler produces a file named ConsoleHelloWorld.exe that you can run on the command line.

You can also create, compile, and run this program in Visual C# .NET, the latest version of Microsoft's integrated development environment. Visual C# .NET is a must for professional C# developers. For certain types of Windows Forms programs—that those that treat the program's window as a form that contains controls such as buttons, text-entry fields, and scroll bars—it's extremely useful. However, it's not strictly necessary. I've found that one of the real pleasures of doing Windows programming in C# with the Windows Forms library is that no separate files are involved. Virtually everything goes in the C# source code file, and everything in that file can be entered with your own fingers and brain.

The following paragraphs describe the steps I took to create the programs in this book using Visual C# .NET. Every sample program in this book is a project, and each project has its own directory of disk storage. In Visual C# .NET, projects are generally grouped into solutions; I created a solution for every chapter in this book. Every solution is also a directory. Projects are subdirectories of the solution directory.

To create a solution, select the menu item File | New | Blank Solution. In the New Project dialog box, select a disk location for this solution and type in a name for the solution. This is how I created solutions for each of the chapters in this book.

When you have a solution loaded in Visual C# .NET, you can create projects in that solution. Select the menu item File | Add Project | New Project. (You can also right-click the solution name in Solution Explorer and select Add | New Project from the context menu.) In the Add New Project dialog box, select a project type of Visual C# Projects. You can choose from several templates. If you want to avoid having Visual C# .NET generate code for you—I personally prefer writing my own code—the template to choose is Empty Project. That's how I created the projects for this book.

Within a project, you can use the Project | Add New Item menu option to create new C# source code files. (Again, you can also right-click the project name in Solution Explorer and select this item from the context menu.) In the Add New Item dialog box, in the Categories list, choose Local Project Items. In the Templates section, choose Code File. Again, if you use that template, Visual C# .NET won't generate code for you.

Regardless of whether you create and compile ConsoleHelloWorld on the command line or in Visual C# .NET, the .exe file will be small, about 3 KB or 4 KB, depending on whether the compiler puts debugging information into it. The executable consists of statements in Microsoft Intermediate Language (MSIL). MSIL has been submitted as a proposed standard to the European Computer
Manufacturer's Association (ECMA), where it is known as the Common Intermediate Language (CIL). When you run the program, the .NET common language runtime compiles the intermediate language to your computer's native machine code and links it with the appropriate .NET DLLs. Currently, you're probably using an Intel-based machine, so the code that the runtime generates is 32-bit Intel x86 machine code.

You can look at MSIL by running the Intermediate Language Disassembler ildasm.exe:

```
ildasm consolehelloworld.exe
```

For documentation on the MSIL instruction set, download the file identified with the acronym "CIL" from [http://msdn.microsoft.com/net/ecma](http://msdn.microsoft.com/net/ecma). Other files on that page may also be useful. You can even write code directly in MSIL and assemble that code using the Intermediate Language Assembler ilasm.exe.

Because programs written in C# are compiled to an intermediate language rather than directly to machine code, the executables are platform independent. Sometime in the future, a .NET runtime environment may be ported to non-Intel machines. If that happens, the executables you're creating today will run on those machines. (Let me add "in theory" so as not to seem hopelessly naïve.)

By using the .NET Framework and programming in C#, you're also creating managed code. This is code that can be examined and analyzed by another program to determine the extent of the code's actions. Managed code is a necessary prerequisite to exchanging binary executables over the Internet.

### Anatomy of a Program

Here's the ConsoleHelloWorld program again.

```
ConsoleHelloWorld.cs
//------------------------------------------------
// ConsoleHelloWorld.cs © 2001 by Charles Petzold
//------------------------------------------------

class ConsoleHelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
    }
}
```

As in C++ and Java (and in many implementations of C), a pair of forward slashes begin a single-line comment. Everything to the right of the slashes doesn't contribute to the compilation of the program. C# also supports multiline comments surrounded by the /* and */ character combinations. One interesting feature of C# is that comments can contain statements in XML (Extensible Markup Language) that can later be formatted and generate documentation of your code. This feature is terrific and I urge you to learn all about it, but I've chosen not to use it in the sample programs in this book.

The entry point to the C# hello-world program is the `Main` function tucked inside the first set of curly brackets. Like C, C++, and Java, C# is case sensitive. Unlike those three languages, the entry point to a C# program is a capitalized `Main` rather than a lowercase `main`. The empty parentheses indicate that the `Main` function has no arguments; the `void` keyword indicates that it returns no value. You can optionally define `Main` to accept an array of character strings as input and to return an integer value. I'll discuss the `public` and `static` keywords later in this chapter. The `public` keyword isn't strictly required here; the program will compile and run fine without it.

The `Main` function is inside a `class` definition. The class is the primary structural and organizational element of object-oriented programming languages such as C#. Very simply, a class is a collection of related code and data. I've given this class a name of `ConsoleHelloWorld`. In this book, I'll
generally (but not always) have one class per source code file. The name of the file will be the name of the class but with a .cs filename extension. This naming convention isn’t required in C#, but the concept was introduced in Java and I like it. Thus, the file that contains the **ConsoleHelloWorld** class definition is **ConsoleHelloWorld.cs**.

`System.Console.WriteLine` appears to be a function call, and indeed it is. It takes one argument, which is a text string, and it displays the text string on the console, in a command-line window, on your vintage teletypewriter, or wherever. If you compile and run this program, the program displays

Hello, world!

and terminates.

That long function name, `System.Console.WriteLine`, breaks down like so:

- **System** is a namespace.
- **Console** is a class defined in that namespace.
- **WriteLine** is a method defined in that class. A method is the same thing that is traditionally called a function, a procedure, or a subroutine.

### C# Namespaces

The namespace is a concept borrowed from C++ and helps ensure that all names used in a particular program or project are unique. It can sometimes happen that programmers run out of suitable global names in a large project or must use third-party class libraries that have name conflicts. For example, you might be coding up a large project in C# and you purchase two helpful class libraries in the form of DLLs from Bovary Enterprises and Karenina Software. Both these libraries contain a class named `SuperString` that is implemented entirely differently in each DLL but is useful to you in both versions. Fortunately, this duplication isn’t a problem because both companies have followed the C# namespace-naming guidelines. Bovary put the code for its `SuperString` class in a namespace definition like so:

```csharp
namespace BovaryEnterprises.VeryUsefulLibrary
{
    class SuperString
    {
    }
}
```

And Karenina did something similar:

```csharp
namespace KareninaSoftware.HandyDandyLibrary
{
    class SuperString
    {
    }
}
```

In both cases, the company name is first, followed by a product name. In your programs that use these libraries, you can refer to the particular `SuperString` class that you need using the fully qualified name

`BovaryEnterprises.VeryUsefulLibrary.SuperString`

or

`KareninaSoftware.HandyDandyLibrary.SuperString`

Yes, it’s a lot of typing, but it’s a solution that definitely works.

This namespace feature would be fairly evil if there weren’t also a way to reduce some of that typing. That’s the purpose of the **using** keyword. You specify a namespace once in the **using** statement, and then you can avoid typing it to refer to classes in that namespace. Here’s an alternative hello-world program for C#.

**ConsoleHelloWithUsing.cs**

```csharp
//----------------------------------------------------
```
using System;

class ConsoleHelloWithUsing
{
    public static void Main()
    {
        Console.WriteLine("Hello, world!");
    }
}

For your project using the two different SuperString classes, the using keyword has an alias feature that helps out:
using Emma = Bovary.VeryUsefulLibrary;
using Anna = Karenina.HandyDandyLibrary;

Now you can refer to the two classes as
Emma.SuperString

and
Anna.SuperString

Consult the C# language reference for more details on the using feature.

The .NET Framework defines more than 90 namespaces that begin with the word System and 5 namespaces that begin with the word Microsoft. The most important namespaces for this book are System itself; System.Drawing, which contains many of the graphics-related classes; and System.Windows.Forms.

Namespaces even allow you to give your own classes names already used in the .NET Framework. The .NET Framework itself reuses some class names. For example, it contains three classes named Timer. These are found in the namespaces System.Timers, System.Threading, and System.Windows.Forms.

What happens to classes that are defined without using a namespace, such as the ConsoleHelloWorld and ConsoleHelloWithUsing classes in my sample programs? Those class names go into a global namespace. This isn't a problem for little self-contained programs like these. However, whenever I define a class in this book that could be useful in someone else's program, I'll put it in the namespace Petzold.ProgrammingWindowsWithCSharp.

Console I/O

Namespaces also play an important role in the organization of the .NET Framework documentation. To find the documentation for the Console class, look in the System namespace. You'll see that WriteLine isn't the only output method in the Console class. The Write method is very similar in that it also displays output to the console. The difference is that WriteLine terminates its output with a carriage return.

There are 18 different definitions of the Write method and 19 different definitions for the WriteLine method, each one with different arguments. These multiple versions of the same method are known as overloads. The compiler can usually figure out which overload a program wants to use by the number and types of the arguments passed to the method.

Here's a program that illustrates three different ways to display the same output.

ConsoleAdder.cs

---

// ConsoleAdder.cs © 2001 by Charles Petzold
using System;

class ConsoleAdder {
    public static void Main() {
        int a = 1509;
        int b = 744;
        int c = a + b;

        Console.Write("The sum of ");
        Console.Write(a);
        Console.Write(" and ");
        Console.Write(b);
        Console.Write(" equals ");
        Console.WriteLine(c);

        Console.WriteLine("The sum of " + a + " and " + b + " equals " + c);

        Console.WriteLine("The sum of {0} and {1} equals {2}", a, b, c);
    }
}

This program displays the following output:
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253

C programmers will be comforted to know that C# supports the familiar int data type and that it doesn't require the := assignment operator of Algol and Pascal.

The first approach the program uses to display the line of output involves separate Write and WriteLine methods, each of which has a single argument. Write and WriteLine can accept any type of variable and will convert it to a string for display.

The second approach uses a technique that C programmers aren't accustomed to but that is familiar to BASIC programmers: string concatenation using the plus sign. C# converts the variables to strings and tacks all the strings together as a single argument to WriteLine. The third method involves a formatting string that has three placeholders, indicated by {0}, {1}, and {2}, for the three other arguments. These placeholders can include additional formatting information. For example, {0:C} displays the number as a currency amount with (depending on the regional settings of the operating system) a dollar sign, commas, two decimal places, and wrapped in a set of parentheses if negative. The placeholder {0:X8} displays the number in hexadecimal, possibly padded with zeros to be eight digits wide. The following table lists some examples of formatting specifications, each applied to the integer 12345.

<table>
<thead>
<tr>
<th>Various Formatting Specifications for the Integer 12345</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format Type</strong></td>
</tr>
<tr>
<td>Currency</td>
</tr>
<tr>
<td>Currency</td>
</tr>
</tbody>
</table>
Various Formatting Specifications for the Integer 12345

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Format Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7</td>
<td>$12,345.0000000</td>
<td></td>
</tr>
<tr>
<td>Decimal</td>
<td>D</td>
<td>12345</td>
</tr>
<tr>
<td></td>
<td>D1</td>
<td>12345</td>
</tr>
<tr>
<td></td>
<td>D7</td>
<td>0012345</td>
</tr>
<tr>
<td>Exponential</td>
<td>E</td>
<td>1.234500E+004</td>
</tr>
<tr>
<td></td>
<td>E1</td>
<td>1.2E+004</td>
</tr>
<tr>
<td></td>
<td>E7</td>
<td>1.2345000E+004</td>
</tr>
<tr>
<td>Fixed point</td>
<td>F</td>
<td>12345.00</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>12345.0</td>
</tr>
<tr>
<td></td>
<td>F7</td>
<td>12345.0000000</td>
</tr>
<tr>
<td>General</td>
<td>G</td>
<td>12345</td>
</tr>
<tr>
<td></td>
<td>G1</td>
<td>1E4</td>
</tr>
<tr>
<td></td>
<td>G7</td>
<td>12345</td>
</tr>
<tr>
<td>Number</td>
<td>N</td>
<td>12,345.00</td>
</tr>
<tr>
<td></td>
<td>N1</td>
<td>12,345.0</td>
</tr>
<tr>
<td></td>
<td>N7</td>
<td>12,345.0000000</td>
</tr>
<tr>
<td>Percent</td>
<td>P</td>
<td>1,234,500.00</td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>1,234,500.0</td>
</tr>
<tr>
<td></td>
<td>P7</td>
<td>1,234,500.0000000</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>X</td>
<td>3039</td>
</tr>
<tr>
<td></td>
<td>X1</td>
<td>3039</td>
</tr>
<tr>
<td></td>
<td>X7</td>
<td>0003039</td>
</tr>
</tbody>
</table>

Even if you don't do much console output in your .NET programming, you'll probably still make use of these formatting specifications in the `String.Format` method. Just as `Console.Write` and `Console.WriteLine` are the .NET equivalents of `printf`, the `String.Format` method is the .NET equivalent of `sprintf`.

**C# Data Types**

I've defined a couple of numbers with the `int` keyword and I've been using strings enclosed in double quotation marks, so you know that C# supports at least two data types. C# actually supports eight integral data types, which are listed here:

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>sbyte</td>
<td>byte</td>
</tr>
<tr>
<td>16</td>
<td>short</td>
<td>ushort</td>
</tr>
<tr>
<td>32</td>
<td>int</td>
<td>uint</td>
</tr>
<tr>
<td>64</td>
<td>long</td>
<td>ulong</td>
</tr>
</tbody>
</table>
C# also supports two floating-point data types, float and double, which implement the ANSI/IEEE Std 754-1985, the IEEE Standard for Binary Floating-Point Arithmetic. The following table shows the number of bits used for the exponent and mantissa of float and double.

<table>
<thead>
<tr>
<th>C# Type</th>
<th>Exponent</th>
<th>Mantissa</th>
<th>Total Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>8</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>11</td>
<td>53</td>
<td>64</td>
</tr>
</tbody>
</table>

In addition, C# supports a decimal data type that uses 128 bits of storage, breaking down into a 96-bit mantissa and a decimal scaling factor between 0 and 28. The decimal data type offers about 28 decimal digits of precision. It's useful for storing and performing calculations on numbers with a fixed number of decimal points, such as money and interest rates. I discuss the decimal data type (and other aspects of working with numbers and mathematics in C#) in more detail in Appendix B.

If you write a literal number such as 3.14 in a C# program, the compiler will assume that it's a double. To indicate that you want it to be interpreted as a float or a decimal instead, use a suffix of f for float or m for decimal.

Here's a little program that displays the minimum and maximum values associated with each of the 11 numeric data types.

```csharp
using System;

class MinAndMax
{
    public static void Main()
    {
        Console.WriteLine("sbyte:   {0} to {1}", sbyte.MinValue, sbyte.MaxValue);
        Console.WriteLine("byte:    {0} to {1}", byte.MinValue, byte.MaxValue);
        Console.WriteLine("short:   {0} to {1}", short.MinValue, short.MaxValue);
        Console.WriteLine("ushort:  {0} to {1}", ushort.MinValue, ushort.MaxValue);
        Console.WriteLine("int:     {0} to {1}", int.MinValue, int.MaxValue);
        Console.WriteLine("uint:    {0} to {1}", uint.MinValue, uint.MaxValue);
        Console.WriteLine("long:    {0} to {1}", long.MinValue, long.MaxValue);
        Console.WriteLine("ulong:   {0} to {1}", ulong.MinValue, ulong.MaxValue);
        Console.WriteLine("float:   {0} to {1}", float.MinValue, float.MaxValue);
        Console.WriteLine("double:  {0} to {1}", double.MinValue, double.MaxValue);
    }
}
```
double.MaxValue);
        Console.WriteLine("decimal: \(0\) to \(1\)", decimal.MinValue,
        decimal.MaxValue);
    }
}

As you'll notice, I've attached a period and the words MinValue and MaxValue onto each data type. These two identifiers are structure fields, and what is going on here will become apparent toward the end of this chapter. For now, let's simply appreciate the program's output:

- sbyte: -128 to 127
- byte: 0 to 255
- short: -32768 to 32767
- ushort: 0 to 65535
- int: -2147483648 to 2147483647
- uint: 0 to 4294967295
- long: -9223372036854775808 to 9223372036854775807
- ulong: 0 to 18446744073709551615
- float: -3.402823E+38 to 3.402823E+38
- double: -1.79769313486232E+308 to 1.79769313486232E+308
- decimal: -79228162514264337593543950335 to 79228162514264337593543950335

C# also supports a bool data type that can take on two and only two values: true and false, which are C# keywords. Any comparison operation (==, !=, <, >, <=, and >=) generates a bool result. You can also define bool data types explicitly. Although you can cast between a bool and an integer (true being converted to 1 and false to 0), this cast must be explicit.

The char data type stores one character, and the string data type stores multiple characters. The char data type is separate from the integer data types and shouldn't be confused or identified with sbyte or byte. For one thing, a char is 16-bits wide (but that doesn't mean you should confuse it with short or ushort either).

The char is 16-bits wide because C# encodes characters in Unicode[^2] rather than ASCII. Instead of the 7 bits used to represent each character in strict ASCII, or the 8 bits per character that have become common in extended ASCII character sets on computers, Unicode uses a full 16 bits for character encoding. This allows Unicode to represent all the letters, ideographs, and other symbols found in all the written languages of the world that are likely to be used in computer communication. Unicode is an extension of ASCII character encoding in that the first 128 characters are defined as in ASCII.

Date types don't need to be defined at the top of a method. As in C++, you can define data types anywhere in the method as you need them.

You can define and initialize a string variable like so:

```csharp
string str = "Hello, World!";
```

Once you've assigned a string to a string variable, the individual characters can't be changed. You can, however, assign a whole new string to the string variable. Strings are not zero-terminated, but you can obtain the number of characters in a string variable using the expression

```csharp
str.Length
```

Length is a property of the string data type, a concept I'll cover later in this chapter. Appendix C contains more information on working with strings in C#.

To define an array variable, use empty square brackets after the data type:

```csharp
float[] arr;
```
The data type of the `arr` variable is an array of floats, but in reality `arr` is a pointer. In C# lingo, an array is a reference type. So is a string. The other data types I've mentioned so far are value types.

When you initially define `arr`, its value is `null`. To allocate memory for the array, you must use the `new` operator and specify how many elements the array has:

```csharp
arr = new float[3];
```

It's actually more common to combine the two statements:

```csharp
float[] arr = new float[3];
```

When you're defining an array, you can also initialize the elements:

```csharp
float[] arr = new float[3] { 3.14f, 2.17f, 100 };;
```

The number of initializers must be equal to the declared size of the array. If you're initializing the array, you can leave out the size:

```csharp
float[] arr = new float[] { 3.14f, 2.17f, 100 };;
```

You can even leave out the `new` operator:

```csharp
float[] arr = { 3.14f, 2.17f, 100 };;
```

Later on in your program, you can reassign the `arr` variable to a `float` array of another size:

```csharp
arr = new float[5];
```

With this call, enough memory is allocated for five `float` values, each of which is initially equal to 0.

You might ask, "What happens to the original block of memory that was allocated for the three `float` values?" There is no `delete` operator in C#. Because the original block of memory is no longer referenced by anything in the program, it becomes eligible for garbage collection. At some point, the common language runtime will free up the memory originally allocated for the array.

As with strings, you can determine the number of elements in an array by using the expression `arr.Length;`

C# also lets you create multidimensional arrays and jagged arrays, which are arrays of arrays.

Unless you need to interface with non-C# code, using pointers in a C# program is rarely necessary. By default, parameters to methods are always passed by value, which means that the method can freely modify any parameter and it won't be changed in the calling method. To change this behavior, you can use the `ref` ("reference") or `out` keywords. For example, here's how you can define a method that modifies a variable passed as an argument:

```csharp
void AddFive(ref int i)
{
    i += 5;
}
```

Here's one that sets a parameter variable:

```csharp
void SetToFive(out int i)
{
    i = 5;
}
```

In the first example, `i` must be set before the call to `AddFive`, and then the value can be changed in the `AddFive` method. In the second example, `i` doesn't have to be set to anything before the method call.

The enumeration plays an important role in C# and the .NET Framework. Many constants throughout the .NET Framework are defined as enumerations. Here's one example from the `System.IO` namespace:
public enum FileAccess
{
    Read = 1,
    Write,
    ReadWrite
}

Enumerations are always integral data types, and the \textit{int} data type by default. If you don't specify an explicit value (as is done for \textit{Read} in this case), the first member is set to the value 0. Subsequent members are set to consecutive values.

You use \textit{FileAccess} in conjunction with several file I/O classes. (Appendix A discusses file I/O in detail.) You must indicate both the enumeration name and the member name separated by a period, as here:

\begin{verbatim}
file.Open(FileMode.CreateNew, FileAccess.ReadWrite)
\end{verbatim}

\textit{FileMode} is another enumeration in the \textit{System.IO} class. If you were to switch around these two enumerations in the \texttt{Open} method, the compiler would report an error. This use of enumerations helps the programmer avoid errors involving constants.


\section*{Expressions and Operators}

One important reference for C programmers is the table that lists the order of evaluation of all the C operations. (It used to be possible to get this table on a T-shirt—printed upside down, naturally, for easy reference.) The equivalent C# table, shown here, is just a little different in the first two lines. It includes a few more operators and excludes the comma operator.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Operator Type} & \textbf{Operators} & \textbf{Associativity} \\
\hline
Primary & () [] f(). x++ y++ new typeof sizeof checked unchecked & Left to right \\
\hline
Unary & + - ! ~ ++ x— x (type) & Left to right \\
\hline
Multiplicative & * / % & Left to right \\
\hline
Additive & + - & Left to right \\
\hline
Shift & << >> & Left to right \\
\hline
Relational & < > <= >= is as & Left to right \\
\hline
Equality & == != & Left to right \\
\hline
Logical AND & \& & Left to right \\
\hline
Logical XOR & ^ & Left to right \\
\hline
Logical OR & | & Left to right \\
\hline
Conditional AND & && & Left to right \\
\hline
Conditional OR & || & Left to right \\
\hline
Conditional & ?: & Right to left \\
\hline
Assignment & += -= *= /= %= <<= >>= |= &= ^= & Right to left \\
\hline
\end{tabular}
\caption{Order of Evaluation in C#}
\end{table}

I want to discuss the two AND and OR operators in some detail because they can provoke some confusion—at least they did when I first encountered them.
Notice that the &, ^, and | operators are termed logical AND, XOR, and OR; in C, these are called the bitwise operators. In C#, the logical AND, XOR, and OR operators are defined for both integral data types and bool. For integral data types, they function as bitwise operators, the same as in C. For example, the expression

0x03 | 0x05

evaluates as 0x07. For bool data types or expressions, they evaluate to bool values. The result of the logical AND operation is true only if both operands are true. The result of the logical XOR is true only if one operand is true and the other is false. The result of the logical OR is true if either of the operands is true.

In C, the && and || operators are known as logical operators. In C#, they're termed conditional AND and OR, and they are defined only for bool data types.

C programmers are accustomed to using the && and || operators in statements like this:

if (a != 0 && b >= 5)

C programmers also know that if the first expression evaluates as false (that is, if a equals 0), then the second expression isn't evaluated. It's important to know this because the second expression could involve an assignment or a function call. Similarly, when you use the || operator, the second expression isn't evaluated if the first expression is true.

In C#, you use the && and || operators in the same way you use them in C. These operators are called the conditional AND and OR because the second operand is evaluated only if necessary.

In C#, you can also use the & and | operators in the same way as && and ||, as in this example:

if (a != 0 & b >= 5)

When you use the & and | operators in this way in C#, both expressions are evaluated regardless of the outcome of the first expression.

The second of the two if statements is also legal in C, and it works the same way as in C#. However, most C programmers would probably write such a statement only in error. That statement simply looks wrong to me, and it sets off a bell in my head because I've trained myself to treat the & as the bitwise AND and && as the logical AND. But in C, the result of a relational or logical expression is an int that has a value of 1 if the expression is true and 0 otherwise. That's why the bitwise AND operation works here.

A C programmer might make the original statement involving the && operator a little more concise by writing it like so:

if (a && b >= 5)

This works fine in C because C treats any nonzero expression as being true. In C#, however, this statement is illegal because the && operator is defined only for bool data types.

Where the C programmer gets into big trouble is using the bitwise AND operator in the shortened form of the expression:

if (a & b >= 5)

If b equals 7, then the expression on the right is evaluated as the value 1. If a is equal to 1 or 3 or any odd number, then the bitwise AND operation yields 1 and the total expression evaluates as true. If a is 0 or 2 or any even number, then the bitwise AND operation yields 0 and the total expression evaluates as false. Probably none of these results are what the programmer intended, and this is precisely why the C programmer has such a violent reaction to seeing bitwise AND and OR operators in logical expressions. (In C#, this statement is illegal because integers and bool values can't be mixed in the logical AND, XOR, and OR statements.)

C# is much stricter than C with regard to casting. If you need to convert from one data type to another beyond what C# allows, the Convert class (defined in the System namespace) provides many methods that probably do what you want. If you need to interface with existing code, the Marshal class (defined in the System.Runtime.InteropServices namespace) contains a method named Copy that lets you transfer between C# arrays and memory areas referenced by pointers.
Conditions and Iterations

C# supports many of the conditional, iteration, and flow control statements used in C. In this section, I'll discuss statements built around the `if`, `else`, `do`, `while`, `switch`, `case`, `default`, `for`, `foreach`, `in`, `break`, `continue`, and `goto` keywords.

The `if` and `else` construction looks the same as in C:
```
if (a == 5)
{
    :
}
else if (a < 5)
{
    :
}
else
{
    :
}
```

In C#, however, the expression in parentheses must resolve to a `bool` data type. This restriction helps the programmer avoid a common pitfall in C of mistakenly using an assignment as the test expression when a comparison is intended:
```
if (a = 5)
```

This statement produces a compilation error in C#, and you'll be thankful that it does.

Of course, no compiler can offer full protection against programmer sleepiness. In one early C# program I wrote, I defined a `bool` variable named `trigger`, but instead of writing the statement
```
if (trigger)
```
I wanted to be a little more explicit and probably intended to type this:
```
if (trigger == true)
```

Unfortunately, I typed this instead:
```
if (trigger = true)
```

This is a perfectly valid statement in C# but obviously didn't do what I wanted.

C# also supports the `do` and `while` statements. You can test a conditional at the top of a block:
```
while (a < 5)
{
    :
}
```

or at the bottom of a block:
```
do
{
    :
}
while (a < 5);
```
The expression must resolve to a `bool` here as well. In the second example, the block is executed at least once regardless of the value of `a`.

The `switch` and `case` construction in C# has a restriction not present in C. In C, you can do this:

```c
switch (a)
{
  case 3:
    b = 7;
    // Fall through isn't allowed in C#.
  case 4:
    c = 3;
    break;
  default:
    b = 2;
    c = 4;
    break;
}
```

In the case where `a` is equal to 3, one statement is executed and then execution falls through to the case where `a` is equal to 4. That may be what you intended, or you may have forgotten to type in a `break` statement. To help you avoid bugs like that, the C# compiler will report an error. C# allows a case to fall through to the next case only when the case contains no statements. This is allowed in C#:

```c
switch (a)
{
  case 3:
  case 4:
    b = 7;
    c = 3;
    break;
  default:
    b = 2;
    c = 4;
    break;
}
```

If you need something more complex than this, you can use the `goto` (described later in this section).

One cool feature of C# is that you can use a string variable in the `switch` statement and compare it to literal strings in the `case` statements:

```c
switch (strCity)
{
  case "Boston":
    :
    break;
  case "New York":
```
Of course, this is exactly the type of thing that causes performance-obsessed C and C++ programmers to cringe. All those string comparisons simply cannot be very efficient. In fact, because of a technique known as string interning (which involves a table of all unique strings used in a program), it's a lot faster than you might think.

The for loop looks the same in C# as in C and C++:

```csharp
for (i = 0; i < 100; i += 3)
{
    :
}
```

As in C++, it's very common for C# programmers to define the iteration variable right in the for statement:

```csharp
for (float f = 0; f < 10.05f; f += 0.1f)
{
    :
}
```

A handy addition is the foreach statement, which C# picked up from Visual Basic. Suppose arr is an array of float values. If you wanted to display all the elements of this array in a single line separated by spaces, you would normally do it like so:

```csharp
for (int i = 0; i < arr.Length; i++)
    Console.Write("{0} ", arr[i]);
```

The foreach statement, which also involves the in keyword, simplifies the operation:

```csharp
foreach (float f in arr)
    Console.Write("{0} ", f);
```

The foreach identifier (named f here) must be assigned a data type in the foreach statement; within the statement or block of statements following foreach, that identifier is read only. As a result, you can't use foreach to initialize the elements of an array:

```csharp
int[] arr = new int[100];
foreach (int i in arr)          // Can't do it!
    i = 55;
```

What's interesting about the foreach statement is that it isn't restricted to arrays. It can be used with any class that implements the IEnumerable interface defined in the System.Collections namespace. Over a hundred classes in the .NET Framework implement IEnumerable. (I'll discuss interfaces briefly later in this chapter and more in Chapter 8.)
The `break` statement normally used with the `switch` and `case` construction will also cause execution flow to jump out of any `while`, `do`, `for`, or `foreach` loop. The `continue` statement jumps to the end of any `while`, `do`, `for`, or `foreach` block; execution flow continues with the next iteration (if any).

And then there's the `goto`:

```c
  goto MyLabel;
  :
MyLabel:
```

The `goto` is useful for getting out of deeply nested blocks and for writing amusingly obscure code. C# also supports a `goto` in the `switch` and `case` construction to branch to another case:

```c
switch (a) {
  case 1:
    b = 2;
    goto case 3;
  case 2:
    c = 7;
    goto default;
  case 3:
    c = 5;
    break;
  default:
    b = 2;
    break;
}
```

You don't need the final `break` at the end of a case if the `goto` is there instead. This feature compensates for not being able to fall through to the next case.

**The Leap to Objects**

In most traditional procedural languages, such as Pascal, Fortran, BASIC, PL/I, C, and COBOL, the world is divided into code and data. Basically, you write code to crunch data.

Throughout the history of programming, programmers have often strived to organize code and data, particularly in longer programs. Related functions might be grouped together in the same source code file, for example. This file might have variables that are used by those isolated functions and nowhere else in the program. And, of course, a formal means to consolidate related `data`, at least, is common in traditional languages in the form of the structure.

Let's suppose you're writing an application and you see that you're going to need to work with dates and, in particular, to calculate day-of-year values. February 2 has a day-of-year value of 33, for example. December 31 has a day-of-year value of 366 in leap years and 365 otherwise. You would probably see the wisdom of referring to the date as a single entity. In C, for example, you can group related data in a structure with three fields:

```c
struct Date
{
  int year;
  int month;
  int day;
}
You can then define a variable of type `Date` like so:

```c
struct Date today;
```

You refer to the individual fields by using a period between the structure variable name and the field name:

```c
today.year = 2001;
today.month = 8;
today.day = 29;
```

But otherwise you can use the variable name (in this case, `today`) to refer to the data as a group. In C, you can also define a structure variable and initialize it in one shot:

```c
struct Date birthdate = { 1953, 2, 2 };
```

To write your day-of-year function, you might begin by writing a little function that determines whether a particular year is a leap year:

```c
int IsLeapYear(int year)
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
```

The `DayOfYear` function makes use of that function:

```c
int DayOfYear(struct Date date)
{
    static int MonthDays[12] = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 };

    return MonthDays[date.month - 1] + date.day +
        ((date.month > 2) && IsLeapYear(date.year));
}
```

Notice that the function refers to the fields of the input structure using the period and the field name.

Here’s a complete working C version of the `Date` structure and related functions.

```c
#include <stdio.h>

struct Date
{
    int year;
    int month;
    int day;
};

int IsLeapYear(int year)
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
```
int DayOfYear(struct Date date)
{
    static int MonthDays[12] = {0, 31, 59, 90, 120, 151,
                                181, 212, 243, 273, 304, 334};

    return MonthDays[date.month - 1] + date.day +
           ((date.month > 2) && IsLeapYear(date.year));
}

int main(void)
{
    struct Date mydate;

    mydate.month = 8;
    mydate.day   = 29;
    mydate.year  = 2001;

    printf("Day of year = %i\n", DayOfYear(mydate));

    return 0;
}

I’ve structured the program with main down at the bottom to avoid forward declarations.

That’s how it’s done in C because the C structure can contain only data types. Code and data are separate and distinct. However, the IsLeapYear and DayOfMonth functions are closely related to the Date structure because the functions are defined only for the Date structure variables. For that reason, it makes sense to consolidate those functions within the Date structure itself. Moving the functions into the structure turns a C program into a C++ program. The C++ version of this program looks like the code on the following page.

CppDateStruct.cpp
//---------------------------------
//-- CppDateStruct.cpp © 2001 by Charles Petzold
//---------------------------------
#include <stdio.h>

struct Date
{
    int year;
    int month;
    int day;

    int IsLeapYear()
    {
        return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 ==
                                   0));
    }

    int DayOfYear()
static int MonthDays[12] = {  0,  31,  59,  90, 120, 151, 181, 212, 243, 273, 304, 334,  

return MonthDays[month - 1] + day + ((month > 2) && IsLeapYear());
};

int main(void)
{
    Date mydate;
    mydate.month = 8;
    mydate.day   = 29;
    mydate.year  = 2001;

    printf("Day of year = %i\n", mydate.DayOfYear());

    return 0;
}

Notice that the total code bulk is smaller. The IsLeapYear and DayOfYear functions no longer have any arguments. They can reference the structure fields directly because they're all part of the same structure. These functions now earn the right to be called methods. Notice also that the struct keyword has been removed in the declaration of the mydate variable in main. It now appears as if Date is a normal data type and mydate is a variable of that type. In object-oriented programming jargon, the mydate variable can now be called an object of type Date, or an instance of Date. Date is sometimes said (by those who have privately practiced saying the word out loud) to be instantiated.

And most important, notice that the DayOfYear method can be called simply by referring to it in the same way you refer to the data fields of the structure: with a period separating the object name and the method name. The more subtle change is a shift of focus: Previously we were asking a function named DayOfYear to crunch some data in the form of a Date structure. Now we're asking the Date structure—which represents a real date on the calendar—to calculate its DayOfYear.

We're now doing object-oriented programming, or at least one aspect of it. We're consolidating code and data into a single unit.

However, in most object-oriented languages, the single unit that combines code and data isn't called a struct. It's called a class. Changing that struct to a class in C++ requires the addition of just one line of code, the keyword public at the top of what is now the definition of the Date class.

CppDateClass.cpp
//------------------------------------------------------------------------
// CppDateClass.cpp © 2001 by Charles Petzold
//------------------------------------------------------------------------
#include <stdio.h>

class Date
{

public:
int year;
int month;
int day;

int IsLeapYear()
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}

int DayOfYear()
{
    static int MonthDays[12] = {0, 31, 59, 90, 120, 151,
                              181, 212, 243, 273, 304, 334};

    return MonthDays[month - 1] + day + ((month > 2) &&
    IsLeapYear());
}

int main(void)
{
    Date mydate;

    mydate.month = 8;
    mydate.day   = 29;
    mydate.year  = 2001;

    printf("Day of year = %i\n", mydate.DayOfYear());

    return 0;
}

In both C++ and C#, a class is very similar to a struct. In both languages, the class isn't exactly the same as the struct, and the class and the struct are different in different ways in the two languages. I'll discuss the C# difference toward the end of this chapter and more in Chapter 3. In C++, all the fields and methods in a struct are public by default; that is, they can be accessed from outside the structure. The fields and methods need to be public because I reference them in main. In a C++ class, all the fields and methods are private by default, and the public keyword is necessary to make them public.

I've done this example in C++ rather than C# because C++ was designed to be compatible with C and thus provides a rather smoother transition from the world of C. Now it's time to do it in C#.

Programming in the Key of C#

The C# version of this program really doesn't look all that much different from the C++ version.

CsDateClass.cs
//------------------------------------------
// CsDateClass.cs © 2001 by Charles Petzold
//------------------------------------------
using System;

int year;
int month;
int day;

int IsLeapYear()
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}

int DayOfYear()
{
    static int MonthDays[12] = { 0, 31, 59, 90, 120, 151,
                                181, 212, 243, 273, 304, 334 };

    return MonthDays[month - 1] + day + ((month > 2) &&
    IsLeapYear());
}

public class Date
{
    public int month;
    public int day;
    public int year;

    public int DayOfYear()
    {
        static int MonthDays[12] = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

        return MonthDays[month - 1] + day + ((month > 2) &&
        IsLeapYear());
    }

    public Date(int month, int day, int year)
    {
        this.month = month;
        this.day = day;
        this.year = year;
    }
}

public class Main
{
    public static void Main()
    {
        Date mydate = new Date(8, 29, 2001);

        Console.WriteLine("Day of year = {0}", mydate.DayOfYear());

        return 0;
    }
}
class CsDateClass
{
    public static void Main()
    {
        Date mydate = new Date();

        mydate.month = 8;
        mydate.day = 29;
        mydate.year = 2001;

        Console.WriteLine("Day of year = (0)", mydate.DayOfYear());
    }
}

class Date
{
    public int year;
    public int month;
    public int day;

    public static bool IsLeapYear(int year)
    {
        return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
    }

    public int DayOfYear()
    {
        int[] MonthDays = new int[] { 0, 31, 59, 90, 120, 151,
                                     181, 212, 243, 273, 304, 334 };

        return MonthDays[month - 1] + day +
               (month > 2 && IsLeapYear(year) ? 1 : 0);
    }
}

One thing I've done, however, is to put the Main method (enclosed in its own class) up at the top of the program and the Date class down at the bottom. I can do this because C# doesn't require forward declarations.

In the C++ version, I defined the Date object like so:
Date mydate;

C# requires a construction like this:
Date mydate = new Date();

As when defining an array, the new keyword performs a memory allocation for the new object of type Date. (I'll discuss the use of parentheses following Date later in this chapter.)

Another change that the C# version requires is the use of the keyword public in front of every field and method in the class that is referenced outside the class. The public keyword is called an access
modifier because it indicates how the fields and methods can be accessed. The other two common alternatives are private and protected, which I'll discuss later in this chapter.

Notice that the IsLeapYear method returns a bool. In the DayOfYear method, I use the conditional operator (?) to generate a value of 1 to add to the day of year for leap years. I could also have cast the bool expression into an int.

Let's get the hang of the jargon: Date is a class. The Date class has five members. The three data members year, month, and day are called fields. The two code members are called methods. The variable mydate is an object of type Date. It's also referred to as an instance of the Date class.

**Static Methods**

I've made another change in converting the C++ version of the program to C#: I added the static modifier to the definition of IsLeapYear and included a year argument to the method. This wasn't a necessary change: if you remove the static keyword and the argument to IsLeapYear, the program will work the same.

But the static modifier is so important in C# and the .NET Framework that I didn't want to delay discussing it another second.

Throughout this chapter, I've been displaying text on the console by specifying the WriteLine method in the Console class:

```csharp
Console.WriteLine(...);
```

However, when calling the DayOfYear method, you specify not the class (which is Date) but mydate, which is an object of type Date:

```csharp
mydate.DayOfYear();
```

You see the difference? In the first case, the class Console is specified; in the second case, the object mydate is specified.

That's the static difference. The WriteLine method is defined as static in the Console class, like so:

```csharp
public static void WriteLine(string value)
```

A static method pertains to the class itself rather than to an object of that class. To call a method defined as static, you must preface it with the name of the class. To call a method not defined as static, you must preface it with the name of an object—an instance of the class in which that method is defined.

This distinction also applies to data members in a class. Any data member defined as static has the same value for all instances of the class. From outside the class definition, the data member must be accessed using the class name rather than an object of that class. Those MinValue and MaxValue fields I used earlier in the MinAndMax program were static fields.

What is the implication of defining IsLeapYear as static? First, you can't call IsLeapYear prefaced with an instance of Date:

```csharp
mydate.IsLeapYear(1997)     // Won't work!
```

You must call IsLeapYear prefaced with the class name:

```csharp
Date.IsLeapYear(1997)
```

Within the class definition (as in the DayOfYear method), you don't need to preface IsLeapYear at all. The other implication is that IsLeapYear must have an argument, which is the year that you're testing. The advantage of defining IsLeapYear as static is that you don't have to create an instance of Date in order to use it. Similarly, you don't need to create an instance of the Console class to use the static methods defined in that class. (Actually, you can't create an instance of Console, and even if you could, you couldn't use it for anything because Console has no nonstatic methods.)

A static method can't call any nonstatic method in the class or use any nonstatic field. That's because nonstatic fields are different for different instances of the class and nonstatic methods return different values for different instances of the class. Whenever you look up something in the .NET Framework reference, you should be alert to see whether or not it's defined as static. It's an
extremely important distinction. Likewise, I'll try to be very careful in this book in indicating when something is defined as static.

Fields can also be defined as static, in which case they're shared among all instances of the class. A static field is a good choice for an array that must be initialized with constant values, such as the MonthDays array in the CsDateClass program. As shown in that program, the array is reinitialized whenever the DayOfYear method is called.

**Exception Handling**

Different operating systems, different graphical environments, different libraries, and different function calls all report errors in different ways. Some return Boolean values, some return error codes, some return NULL values, some beep, and some crash the system.

In C# and the .NET Framework, an attempt has been made to uniformly use a technique known as structured exception handling for reporting all errors.

To explore this subject, let's begin by setting the month field of the Date object in the CsDateClass program to 13:

```csharp
mydate.month = 13;
```

Now recompile and run the program. If a dialog box comes up about selecting a debugger, click No. You'll then get a message on the command line that says this:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
    at Date.DayOfYear()
    at CsDateClass.Main()
```

If you've compiled with debug options on, you'll get more explicit information that indicates actual line numbers in the source code. In either case, however, the program will have been prematurely terminated.

Notice that the message is accurate: the index to the MonthDays array was truly out of range. In C, a problem like this might result in some other kind of error, such as a stack overflow. C# checks whether an array index is valid before indexing the array. The program responds to an anomalous index by a simple process that's called throwing (or raising) an exception.

It's possible—and in fact very desirable—for programs themselves to know when exceptions are occurring and to deal with them constructively. When a program checks for exceptions, it is said to catch the exception. To catch an exception, you enclose statements that may generate an exception in a try block and statements that respond to the exception in a catch block. For example, you can try putting the following code in the CsDateClass program with the bad date:

```csharp
try
{
    Console.WriteLine("Day of year = {0}", mydate.DayOfYear());
}
catch (Exception exc)
{
    Console.WriteLine(exc);
}
```

*Exception* is a class defined in the System namespace, and *exc* is an object of type *Exception* that the program is defining. This object receives information about the exception. In this example, I've chosen simply to pass *exc* as an argument to *Console.WriteLine*, which then displays the same block of text describing the error I showed you earlier. The difference, however, is that the program isn't prematurely terminated and could have handled the error in a different way, perhaps even a graceful way.
A single line of code can cause several types of exceptions. For that reason, you can define different catch blocks:

```csharp
try
{
    ...
}
catch (NullReferenceException exc)
{
    ...
}
catch (ArgumentOutOfRangeException exc)
{
    ...
}
catch (Exception exc)
{
    ...
}
```

Notice that the most generalized exception is at the end.

You can also include a finally block:

```csharp
try
{
    ...
}
catch (Exception exc)
{
    ...
}
finally
{
    ...
}
```

Regardless of whether or not an exception occurs, the code in the finally block is executed following the code in the catch block (if an exception occurred) or the code in the try block (if there was no exception). You can put cleanup code in the finally block.

You might ask, Why do I need the finally block? Why can't I simply put my cleanup code after the catch block? That's certainly possible. However, you could end your try or catch blocks with goto statements. In that case, the code in the finally block would be executed anyway, before the goto occurred.

It's also possible to leave out the catch block:

```csharp
try
{
    ...
}
finally
```
In this case, you'd get the dialog box about a debugger and a printed version of the exception (the same as displaying it with `Console.WriteLine`), and then the code in the `finally` clause would be executed and the program would proceed normally.

**Throwing Exceptions**

What still bothers me in this particular case is that we really haven't gotten to the root of the problem. The `DayOfYear` method is throwing an exception because the index to the `MonthDays` array is out of bounds. But the real problem occurs earlier in the program, with this statement that I told you to put in the program:

```csharp
mydate.month = 13;
```

Once this statement is executed, you're dealing with a `Date` object that contains an invalid date. That's the real problem. It just so happens that `DayOfYear` was the first method that had a bad reaction to this problem. But suppose you put the following statement in the program:

```csharp
mydate.day = 47;
```

The `DayOfYear` method goes right ahead and calculates a result despite the fact that it's dealing with a bogus date.

Is there a way for the class to protect itself against the fields being set to invalid values by a program using the class? The easy way is by marking the fields as `private` rather than `public`:

```csharp
private int year;
private int month;
private int day;
```

The `private` modifier makes the three fields accessible only from methods inside the `Date` class definition. In fact, in C#, the `private` attribute is the default, so you only need to remove the `public` attribute to make this change:

```csharp
int year;
int month;
int day;
```

Of course, this change creates its own problem: How is a program that uses the `Date` class supposed to set the values of the year, month, and day?

One solution that might occur to you is to define methods in the `Date` class specifically for setting these three fields and also for getting the values once they're set. For example, here are two simple methods for setting and getting the private `month` field:

```csharp
public void SetMonth (int month)
{
    this.month = month;
}

public int GetMonth ()
{
    return month;
}
```

Notice that both these methods are defined as `public`. Notice also that I've given the name of the argument variable in `SetMonth` the same name as the field! If you do this, the field name needs to be prefaced with the word `this` and a dot. Inside a class, the keyword `this` refers to the instance of the class that's calling the method. The `this` keyword is invalid in static methods.
Here's a version of `SetMonth` that checks for proper month values:

```java
public void SetMonth (int month)
{
    if (month >= 1 && month <= 12)
        this.month = month;
    else
        throw new ArgumentOutOfRangeException("Month");
}
```

And there's the syntax for throwing an exception. I've chosen `ArgumentOutOfRangeException` because that one most closely identifies the problem. The `new` keyword creates a new object of type `ArgumentOutOfRangeException`. That object is what the `catch` block gets as a parameter. The argument to `ArgumentOutOfRangeException` is a text string that identifies the parameter causing the problem. This text string is included along with the other information about the error if you choose to display it.

C# has a better alternative to `Get` and `Set` methods. Whenever you're on the verge of writing methods that begin with the words `Get` or `Set`—indeed, whenever you're on the verge of writing any method that returns information about an object and that doesn't require an argument—you should think of a C# feature known as the `property`.

### Getting and Setting Properties

As you've seen, C# classes can contain data members that are called `fields` and code members that are called `methods`. C# classes can also contain other code members, called `properties`, that are extremely important in the .NET Framework.

Properties seem to blur the distinction between code and data. To a program using the class, properties look like data fields, and they can often be treated like data fields. Within a class, however, a property is definitely code. In many cases, a public property provides other classes access to a private field in the class. The property has the advantage over a field of being able to perform validity checks.

Some C# programmers (like myself) give private fields names that begin with lowercase letters and public properties names that begin with uppercase letters. Here's a simple definition of a `Month` property that provides access to the private `month` field:

```java
public int Month
{
    set
    {
        month = value;
    }
    get
    {
        return month;
    }
}
```

A program using a class with such a property refers to the property in the same way as it might refer to a field:

```java
mydate.Month = 7;
```

or

```java
Console.WriteLine(mydate.Month);
```

or
mydate.Month += 2;

The final example increases the Month property by 2. See how much cleaner this syntax is than an equivalent statement using those SetMonth and GetMonth methods we toyed with earlier:

mydate.SetMonth(mydate.GetMonth() + 2); // Good riddance!

Let's examine the property definition in detail: The public keyword indicates that this property is accessible from outside the class. The int data type indicates that the property is a 32-bit integer. The property itself is named Month.

Within the body of the property are two accessors, named set and get. You don't have to include both. Many properties have only public get accessors, in which case the set accessor is either not defined at all or defined as private. Such properties are known as read-only properties. It's also possible to have a property with a set accessor and no get accessor, but these are much rarer.

Within the definition of the set accessor, the special word value refers to the value that property is being set to by a statement such as this:

mydate.Month = 7;

A get accessor always contains a return statement to return a value to the program using the property.

Here's a program that defines Year, Month, and Day properties and implements validity checking in the set accessors.

CsDateProperties.cs
//--
// CsDateProperties.cs © 2001 by Charles Petzold
//--
using System;

class CsDateProperties
{
    public static void Main()
    {
        Date mydate = new Date();

        try
        {
            mydate.Month = 8;
            mydate.Day   = 29;
            mydate.Year  = 2001;

            Console.WriteLine("Day of year = {0}", mydate.DayOfYear);
        }
        catch (Exception exc)
        {
            Console.WriteLine(exc);
        }
    }
}

class Date
{
// Fields

int year;
int month;
int day;
static int[] MonthDays = new int[] { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

// Properties

public int Year
{
    set
    {
        if (value < 1600)
            throw new ArgumentOutOfRangeException("Year");
        else
            year = value;
    }
    get
    {
        return year;
    }
}

public int Month
{
    set
    {
        if (value < 1 || value > 12)
            throw new ArgumentOutOfRangeException("Month");
        else
            month = value;
    }
    get
    {
        return month;
    }
}

public int Day
{
    set
    {
        if (value < 1 || value > 31)
            throw new ArgumentOutOfRangeException("Day");
        else
            day = value;
    }
}
public int DayOfYear
{
    get
    {
        return MonthDays[month - 1] + day +
            (month > 2 && IsLeapYear(year) ? 1 : 0);
    }
}

// Method
public static bool IsLeapYear(int year)
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}

I've left in the try and catch code so that you can experiment with invalid dates. Notice that I've also set a minimum of 1600 on the Year property. The IsLeapYear method doesn't make much sense for dates earlier than that. One problem that still remains is that the individual properties don't test for consistency. You can set a date of February 31, for example. Such consistency checking would impose restrictions on the order in which the properties were set, so I'm going to let that go.

I've also changed DayOfYear from a method to a read-only property, just because the value seemed to me more like a property of a date rather than a method. Sometimes it's hard to determine whether something should be a method or a property. The only obvious rule is, If it has an argument, it's gotta be a method.

Constructors

Back in the C version of the program, I mentioned briefly that you can initialize the fields of a structure when you define a structure variable:

struct Date birthdate = { 1953, 2, 2 };

I didn't really pursue this throughout the various versions, however. It's possible to initialize a C++ class or struct in such a way, but such an initialization is dependent on the number of fields in a struct or class and the ordering of those fields, and it's probably not a good idea. In C#, it's not allowed. But it certainly would be nice to do something like this in C#.

Another issue: The previous version of the C# program implements validity checking in all the set accessors of its properties. However, there's still a situation in which the class has an invalid date, and that's when the object is first created:

Date mydate = new Date();

You can solve both these problems with something called a constructor. A constructor is a method in the class that is run when an object of that class is created. If you look at the expression following the word new in

Date mydate = new Date();

you'll see what seems to be a method call with no arguments. That's exactly what it is! It's a call to the default constructor of Date. Every class has a default constructor that exists whether or not you
explicitly define it. But if you explicitly define a default constructor in the *Date* class, you can make sure that the *Date* object always has a valid date.

It's also possible to define constructors that have one or more arguments. In the *Date* class, you might want to define a constructor with three arguments that initializes a *Date* object with a particular date. Such a constructor would allow you to create a *Date* object like so:

```java
Date birthdate = new Date(1953, 2, 2);
```

In the class, the constructor looks a lot like a method except that it has the same name as the class in which it is defined and it has no return type. If you put a return type on a constructor or if you define any other method without a return type, you'll get an error from the compiler. This is good because it lets you know whether you’ve typed the class name wrong when defining the constructor.

Here's a simple approach to a constructor that includes date arguments:

```java
public Date(int year, int month, int day)
{
    this.year = year;
    this.month = month;
    this.day = day;
}
```

But it doesn't use all the error checking we've implemented in the properties. A better approach is for the constructor to set the properties rather than the fields:

```java
public Date(int year, int month, int day)
{
    Year = year;
    Month = month;
    Day = day;
}
```

In fact, you can do more than this. You can actually perform consistency checks among the three values in the constructor.

What about the default constructor? It's common for classes to define a default constructor that sets the object to a value of 0, or something more or less equivalent to a 0 value. For the *Date* class, that probably means the date January 1, 1600 because that's the earliest date allowed. Here's the new version of the program.

```java
CsDateConstructors.cs
//****************************************************************************
// CsDateConstructors.cs © 2001 by Charles Petzold
//****************************************************************************
using System;

class CsDateConstructors
{
    public static void Main()
    {
        try
        {
            Date mydate = new Date(2001, 8, 29);

            Console.WriteLine("Day of year = " + mydate.DayOfYear);
        }
    }
}
```
catch (Exception exc) {
    Console.WriteLine(exc);
}
}
}

class Date {

    // Fields
    int year;
    int month;
    int day;

    static int[] MonthDays = new int[] { 0, 31, 59, 90, 120, 151,
                                           181, 212, 243, 273, 304, 334 }; // Fields

    // Constructors
    public Date() {
        Year = 1600;
        Month = 1;
        Day = 1;
    }

    public Date(int year, int month, int day) {
        if ( (month == 2 && IsLeapYear(year) && day > 29) ||
            (month == 2 && !IsLeapYear(year) && day > 28) ||
            ((month == 4 || month == 6 ||
               month == 9 || month == 11) && day > 30))
            throw new ArgumentOutOfRangeException("Day");
        } else
        {
            Year = year;
            Month = month;
            Day = day;
        }
    }

    // Properties
    public int Year {
        set {
            if (value < 1600)
throw new ArgumentOutOfRangeException("Year");
else
    year = value;
}
get
{
    return year;
}
}

public int Month
{
    set
    {
        if (value < 1 || value > 12)
            throw new ArgumentOutOfRangeException("Month");
        else
            month = value;
    }
    get
    {
        return month;
    }
}

public int Day
{
    set
    {
        if (value < 1 || value > 31)
            throw new ArgumentOutOfRangeException("Day");
        else
            day = value;
    }
    get
    {
        return day;
    }
}

public int DayOfYear
{
    get
    {
        return MonthDays[month - 1] + day +
            (month > 2 && IsLeapYear(year) ? 1 : 0);
    }
}
public static bool IsLeapYear(int year)
{
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}

Instances and Inheritance

There may come a time when you're using a class and you think, "This class is pretty good, but it'd be even better if it did ..." something or other. If you have the source code to the class, you could simply edit it, put the new method in, recompile, and go. But you may not have the source code. You may have access only to a compiled version of the class implemented in a DLL.

Or maybe there's something the class does that you'd like it to do a little differently. But you're using the class as is in other applications, and it's fine there. It just needs this change for your new application, and you'd prefer not to mess around with the source code for the original version.

That's why object-oriented languages like C# implement a feature known as inheritance. You can define a new class based on an existing class. It's said that you inherit from an existing class, or subclass an existing class. The new class need contain only the new stuff. All classes in C# and the .NET Framework inherit from a class named Object or from a class inherited from Object. It's also said that all classes ultimately derive from Object.

Let's create a new class named DatePlus that inherits from Date. DatePlus is going to have a new property named DaysSince1600. And because it implements such a property, DatePlus can calculate the difference in days between two dates.

Here's the program that defines the DatePlus class.

CsDateInheritance.cs
//------------------------------------------------
// CsDateInheritance.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;

class CsDateInheritance
{
    public static void Main()
    {
        DatePlus birth = new DatePlus(1953, 2, 2);
        DatePlus today = new DatePlus(2001, 8, 29);

        Console.WriteLine("Birthday = {0}", birth);
        Console.WriteLine("Today = {0}", today);
        Console.WriteLine("Days since birthday = {0}". today - birth);
    }
}

class DatePlus : Date
{
    public DatePlus() {}
    public DatePlus(int year, int month, int day): base(year, month, day) {}
public int DaysSince1600
{
    get
    {
        return 365 * (Year - 1600) +
            (Year - 1597) / 4 -
            (Year - 1601) / 100 +
            (Year - 1601) / 400 + DayOfYear;
    }
}

public override string ToString()
{
    string[] str = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",

    return String.Format("{0} {1} {2}", Day, str[Month - 1], Year);
}

public static int operator -(DatePlus date1, DatePlus date2)
{
    return date1.DaysSince1600 - date2.DaysSince1600;
}

When you compile this program, you must compile it along with the CsDateConstructors.cs file, which is the most recent file that implements the Date class. Because you now have two classes that have a Main method, you must tell the compiler which class contains the Main method you want to use as the program’s entry point.

If you’re compiling on the command line, you need to use

csc CsDateConstructors.cs CsDateInheritance.cs /main:CsDateInheritance

Watch out for uppercase and lowercase here. You can type the filename arguments in whatever case you want, but the /main argument refers to a class, and the case must match the class name exactly as defined in the file. If you’re using Visual C# .NET, you need to add CsDateConstructors.cs to the CsDateInheritance project. To do this, choose Add Existing Item from the Project menu. When you select CsDateConstructors.cs in the Add Existing Item dialog box, click the arrow next to the Open button and select Link File. Selecting this option avoids having to make a copy of the CsDateConstructors.cs file and also avoids problems that occur when you change one version of the file but not the other.

Notice the first line of the DatePlus definition:

class DatePlus: Date

That means DatePlus inherits from Date. DatePlus doesn’t need to do anything special in its constructors. For that reason, it defines the default constructor with an empty body:

    public DatePlus() {}

Whenever you create an instance of a class, all the default constructors of all the objects that the class derives from are called, starting with the default constructor for Object and ending with the default constructor for the class you’re creating an object of.

The same isn’t true of nondefault constructors. The constructor with three arguments doesn’t need to do anything special in DatePlus, but you need to include it and you need to explicitly call the
constructor in the base class, which is the class that DatePlus inherits from, namely Date. Here's the syntax:

```csharp
public DatePlus(int year, int month, int day): base(year, month, day) {}
```

Again, the constructor does nothing special in DatePlus, so the body is empty.

The DatePlus class implements two other neat features besides the DaysSince1600 property. First, DatePlus defines the minus operator (−) for objects of this class. This is called overloading the operator. The minus operator is normally defined only for numbers, but here we're saying you can use it for dates as well. The body of this operator overload is fairly simple: it just subtracts one DaysSince1600 property from another.

So if you define two DatePlus objects as

```csharp
DatePlus birth = new DatePlus(1953, 2, 2);
DatePlus today = new DatePlus(2001, 8, 29);
```

you can find the difference in days simply by using the expression

```csharp
today - birth
```

Notice that I didn't implement an override of the plus operator in this class. It wouldn't make sense to add two dates together. However, I could have implemented the addition of a date and an integer to yield a new date. But I would also need some code to convert a new day-since-1600 value back to a date. Implementing comparison operators (≤, ≥, <, and >) would be fairly easy, though.

I mentioned earlier that all classes ultimately derive from Object. The Object class implements a method named ToString that's intended to convert an object into a human-readable text string. We've actually already made use of ToString. Whenever you concatenate a numeric variable with a text string, the ToString method of the variable is automatically called. Whenever you pass an object to Console.WriteLine, the ToString method of the object is called.

However, the default behavior of the ToString method in Object is to return the name of the class, for example, the text string "DatePlus". But that's OK, because any class that derives from Object (and that means any class defined in C#) can override the ToString method in Object by providing its own. The DatePlus class implements its own ToString method and uses the static method String.Format to format the date into a text string. You can then pass a DatePlus object to Console.WriteLine and get a formatted date. The output of the CsDataInheritance program looks like this:

```
Birthday = 2 Feb 1953
Today = 29 Aug 2001
Days since birthday = 17740
```

We're now ready to look at access modifiers in more detail. If you define a field, property, or method as private, it is visible and accessible only from within the class. If you define a field, property, or method as public, it is visible and accessible from other classes. If you define a field, property, or method as protected, it is visible and accessible only from within the class and in any class that inherits from the class.

The ToString method in the Object class is defined with the modifier virtual. A method defined as virtual is intended to be overridden by classes that derive from the class. A method that overrides the virtual method uses the override modifier to indicate that it wants to replace a method with its own version. The override modifier is required so that you won't make the mistake of accidentally overriding a virtual method when you didn't intend to.

A class can also override a method that isn't defined as virtual. In that case, the new method must include the modifier new.

Besides ToString, the Object class also defines several other methods, including GetType. GetType returns an object of type Type, a class defined in the System namespace. The Type class allows you to obtain information about the object, including its methods, properties, and fields. The C# typeof operator also returns an object of type Type. The difference is that GetType is applied to an object while typeof is applied to a class. In the Main method in CsDataInheritance, the expression

```csharp
today.GetType() == typeof(DatePlus)
```
A Bigger Picture

The documentation of the class libraries in the .NET Framework is organized by namespace. Each namespace is a logical grouping of classes (and such) and is implemented in a particular DLL.

Within each namespace you’ll see five types of items. These are the only five types of items that can be defined on the external level in C#:
- A class, which we’ve already encountered.
- A struct is very similar to a class.
- An interface is similar to a class or struct but defines only method names rather than bodies. (Chapter 8 has an example of an interface.)
- An enumeration is a list of constants with predefined integer values.
- A delegate is a prototype of a method call.

The class and the struct are ostensibly very similar in C#. A class, however, is a reference type, which means that the object is really a pointer into an allocated block of memory. A struct is a value type, more like a regular numeric variable. I’ll discuss the difference in more detail in Chapter 3. I’ll talk about the delegate in the next chapter; it’s most commonly used in conjunction with events.

Some classes in the .NET Framework contain static methods and properties that you’ll call by specifying the class name and the method (or property) name. Some classes in the .NET Framework you’ll instantiate in your Windows Forms applications. And some classes in the .NET Framework you’ll inherit in your applications.

Within a class or a struct you’ll find the following members:
- Fields, which are objects of specific types
- Constructors, which are executed when an object is created
- Properties, which are blocks of code with set and get accessors
- Methods, which are functions that accept arguments and return values
- Operators, which implement standard operators such as + and − defined for the object, or casts
- Indexers, which allow the object to be referenced like an array
- Events, which I’ll discuss in the next chapter
- Other embedded classes, structures, interfaces, enumerations, or delegates

Early in this discussion of C#, I covered numeric types and string types supported by the language. All the basic types in C# are implemented as classes or structures in the System namespace. The int data type, for example, is an alias for the Int16 structure. Rather than define an int as

```csharp
int a = 55;
```

you can use

```csharp
System.Int16 a = 55;
```

These two statements are functionally identical, which is why you sometimes see strings in C# defined like so:

```csharp
string str = "Hello, world!";
```

And sometimes with a capitalized String data type:

```csharp
String str = "Hello, world!";
```

The appearance of uppercase and lowercase types in these two statements doesn't mean that C# is sometimes case insensitive. The capitalized String refers to the String class in the System namespace. If you don’t have a using statement for the System namespace, you’d need to use

```csharp
System.String str = "Hello, world!";
```

if you want to use String rather than string.

Here’s a table showing how the C# types correspond to classes and structures in the System namespace:

| C# Data Types Aliases |
Signed | Unsigned
---|---
.NET Type | C# Alias | .NET Type | C# Alias
System.Object | object | System.Enum | enum
System.String | string | System.Char | char
System.SByte | sbyte | System.Byte | byte
System.Int16 | short | System.UInt16 | ushort
System.Int32 | int | System.UInt32 | uint
System.Int64 | long | System.UInt64 | ulong
System.Single | float | System.Double | double
System.Decimal | decimal | System.Boolean | bool

Because basic types are classes and structures, they can have fields, methods, and properties. This is how the `Length` property can return the number of characters in a `string` object and how the numeric data types can have fields named `MinValue` and `MaxValue`. Arrays support properties and methods implemented in the `System.Array` class.

**Naming Conventions**

Throughout the remainder of this book, I'll use naming conventions that are based somewhat on the .NET Framework and somewhat on a system called Hungarian notation, named in honor of legendary Microsoft programmer Charles Simonyi.

For class names, property names, and event names that I define, I'll use *Pascal casing*. This system is a mixture of uppercase and lowercase beginning with a capital and possibly containing embedded capitals.

For fields, variables, and objects I define, I'll use *camel casing*. The first letter is lowercase but the name may include uppercase letters. (The uppercase letters are the camel's humps.)

For variables of the standard types, I'll use a lowercase prefix on the variable name that indicates the type of the variable. Here are the prefixes I use in this book:

<table>
<thead>
<tr>
<th>Date Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>b</td>
</tr>
<tr>
<td>byte</td>
<td>by</td>
</tr>
<tr>
<td>short</td>
<td>s</td>
</tr>
<tr>
<td>int</td>
<td>i, x, y, cx, cy</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
</tr>
<tr>
<td>float</td>
<td>f</td>
</tr>
<tr>
<td>char</td>
<td>ch</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>object</td>
<td>obj</td>
</tr>
</tbody>
</table>

The `x` and `y` prefixes indicate coordinate points. The `cx` and `cy` prefixes indicate widths and heights. (The `c` stands for `count`.)

For objects created from classes, I'll use a lowercase version of the class name as a prefix, sometimes abbreviated. For example, an object of type `Point` may be called `ptOrigin`. Sometimes the program will create only one object of a particular class, so the object can be the same as the class name but in lowercase. For example, an object of type `Form` will be named `form`. An object of type `PaintEventArgs` will be named `pea`. 
Any array variable will be prefixed with an a before any other prefix.

**Beyond the Console**

In fall 1985, Microsoft released the first version of Windows. At the same time, Microsoft also released the Windows Software Development Kit (SDK), which showed programmers how to write Windows applications in C.

The original hello-world program in the Windows 1.0 SDK was a bit of a scandal. HELLO.C was about 150 lines long, and the HELLO.RC resource script had another 20 or so more lines. Granted, the program created a menu and displayed a simple dialog box, but even so, leaving out those amenities still left about 70 lines of code. Veteran C programmers often curled up in horror or laughter when first encountering the Windows hello-world program.

In a sense, the whole history of new programming languages and class libraries for Windows has involved the struggle to reduce the Windows hello-world program down to something small, sleek, and elegant.

Let's see how Windows Forms fares in this respect.
Chapter 2: Hello, Windows Forms

Overview

The programs shown in the previous chapter were not, of course, Windows programs. Those programs didn't create their own windows, didn't draw any graphics, and knew nothing about the mouse. All the user input and output came through a class named Console. It's time to move on. For the remainder of this book, the Console class won't be entirely forgotten, but it will be relegated to relatively mundane chores such as logging and primitive debugging.

Which raises the question: What exactly is the difference between a console application and a Windows application? Interestingly enough, the distinction isn't quite as clear-cut as it used to be. A single application can have elements of both. It can start out as a console application and then become a Windows application, and go back to being a console application again. A Windows application can also display console output with impunity. A console application can display a Windows message box to report a problem and then resume console output when the user dismisses that message box.

To the C# compiler, the difference between a console application and a Windows application is a compiler switch named target (which can be abbreviated t). To create a console application, use the switch

/target:exe

That's the default if you specify no target switch. To create a Windows executable, use

/target:winexe

The target switch can also indicate a library or a module. In Microsoft Visual Studio .NET, you use the project Property Pages dialog box. In the General Common Properties section, set the Output Type to either Console Application or Windows Application.

This compiler switch doesn't do anything very profound. It really only sets a flag in the executable file that indicates how the program is to be loaded and run. If an executable is flagged as a Console Application and is started from Windows, the Windows operating system creates a Command Prompt window that launches the program and displays any console output from the program. If the console application is started from within the Command Prompt window, the MS-DOS prompt doesn't return until the program terminates. If the executable is flagged as a Windows Application, no Command Prompt window is created. Any console output from the program goes into the bit bucket. If you start such a program from the Command Prompt window, the MS-DOS prompt appears again right after the program is launched. The point is this: nothing bad happens if you compile a Windows Forms application as a console application!

One thing to keep in mind is that the Command Prompt window behaves differently depending on whether you're running in release mode or debug mode. If you're in release mode, you'll see the standard "Press any key to continue" message in the console when a program ends. At that point, you'll still be able to see any output sent to the console and you can then dismiss the console window when you're done viewing the output. If you're in debug mode and you start the program from Windows, the console window will disappear without warning as soon as the program ends. You'll need to view any output to the console before shutting down the program.

All the Visual Studio .NET project files that accompany the programs from this book specify that the programs are console applications. That's why when you execute these programs, a Command Prompt window comes up first. That console is to your advantage: if you ever need to see what's going on inside one of these programs, you can simply stick Console.Write or Console.WriteLine statements anywhere in any program in this book. (Although as I mentioned, you won't have an opportunity to view these statements if you run in debug mode and end the program. In such cases, you'll also want to be sure not to put the Write or WriteLine statements in the code to display after the program window has shut down.) There are very few mysteries in life that can't be cleared up with a couple Console.WriteLine statements. (There's also a Debug class in the System.Diagnostics namespace that provides alternatives to using the Console class for this purpose.)

Of course, I wouldn't send a Windows program compiled as a console application out into the nondeveloper marketplace. Users might get upset seeing a Command Prompt window popping up
(unless they are familiar with UNIX and UNIX-like environments). But it's only a compiler switch, and that can be changed at any time.

The real difference between a console application and a Windows application is the way in which the program gets user input. A console application gets keyboard input through the `Console.Read` or `Console.ReadLine` methods; a Windows Forms application gets keyboard (and other) input through events, a subject we'll be studying for much of this book.

I created the projects for this chapter in Visual Studio .NET in much the same way I created the projects in Chapter 1. I specified that the project was a Visual C# Project but that it was an Empty Project. When I created a program in the project, I used the Add New Item menu option and specified a Local Project Item and a Code File. This process dissuades Visual Studio .NET from generating code for you. In this book, you and I will be writing our own code.

However, the C# compiler needs access to some additional DLLs that are part of the .NET Common Language Runtime (CLR) environment. If you're running the C# compiler on the command line, you need to include the `reference` (abbreviated `r`) compiler switch:

```
```

You'll also need to specify these three files in Visual Studio .NET. In Solution Explorer, right-click on the References item underneath the project name and select Add Reference from the context menu. (You can also select the Add Reference item from the Project menu.) Select these three items from the list in the dialog box that you're presented with:

- `System.dll`
- `System.Drawing.dll`
- `System.Windows.Forms.dll`

If you have multiple projects grouped in a Visual Studio .NET solution (as the projects for this book are organized), you need to specify these files only for the first project. You can then select these three files in Solution Explorer as they are listed in the References section of one project and drag them to the References section of each subsequent project.

### The MessageBox

At the beginning of the chapter, I mentioned message boxes. Let's take a look at a short but authentic Windows Forms program that displays our favorite two words of deathless prose.

**MessageBoxHelloWorld.cs**

```csharp
//---------------------------------------------------
// MessageBoxHelloWorld.cs © 2001 by Charles Petzold
//---------------------------------------------------
class MessageBoxHelloWorld
{
    public static void Main()
    {
    }
}
```

This program is quite similar to the original `ConsoleHelloWorld` program in Chapter 1. It has a class (`MessageBoxHelloWorld`), a method in that class named `Main` that's the entry point to the program, and a single executable statement that's really not too much longer than the console equivalent. That long function name breaks down like so:

- `System.Windows.Forms` is a namespace.
- `MessageBox` is a class in that namespace.
- `Show` is a static method in the `MessageBox` class.

Because `Show` is a static method, it must be prefaced with the class name and not an object created from that class, just like the `WriteLine` method of the `Console` class. Here's what the output of this program looks like:
When you press the OK button, the message box goes away, the *Show* method returns, and the program terminates.

*System.Windows.Forms* is a gigantic namespace that contains around 200 classes and 100 enumerations as well as about 41 delegates, 7 interfaces, and 4 structures. Together with *System* and *System.Drawing*, it is the most important namespace in this book. Customarily, you'll put the statement

```csharp
using System.Windows.Forms;
```

at the top of your Windows Forms programs; you can then refer to the static *Show* method of *MessageBox* simply as:

```csharp
MessageBox.Show("Hello, world!");
```

You've probably seen plenty of message boxes when you've worked with Windows. Message boxes always contain a brief message to the user and let the user respond by clicking a button, sometimes one of two or three buttons. Optionally, the message can be adorned with an icon and a descriptive caption. Programmers can also use message boxes for debugging purposes because they offer a quick way to display text information and temporarily suspend the program.

*MessageBox* is derived from *Object* and thus inherits a few methods implemented by *Object*. The only method *MessageBox* itself implements is *Show*. It's a *static* method and exists in 12 different versions. Here are 6 of them:

**MessageBox Show Methods (selection)**

```csharp
DialogResult Show(string strText)
DialogResult Show(string strText, string strCaption)
DialogResult Show(string strText, string strCaption,
                  MessageBoxButtons mbb)
DialogResult Show(string strText, string strCaption,
                  MessageBoxButtons mbb, MessageBoxIcon mbi)
DialogResult Show(string strText, string strCaption,
                  MessageBoxButtons mbb, MessageBoxIcon mbi,
                  MessageBoxDefaultButton mbdb)
DialogResult Show(string strText, string strCaption,
                  MessageBoxButtons mbb, MessageBoxIcon mbi,
                  MessageBoxDefaultButton mbdb, MessageBoxOptions mbi)
```

The other six overloaded *Show* methods are used in connection with Win32 code. The text you specify in the message box caption is typically the name of the application. Here's an alternative *MessageBox.Show* call for our first Windows Forms program:

```csharp
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld");
```
When you don't use the second argument, no text appears in the caption bar.

You can choose one of the following enumeration values to indicate the buttons that appear on the message box:

<table>
<thead>
<tr>
<th><strong>MessageBoxButtons Enumeration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>OK</td>
</tr>
<tr>
<td>OKCancel</td>
</tr>
<tr>
<td>AbortRetryIgnore</td>
</tr>
<tr>
<td>YesNoCancel</td>
</tr>
<tr>
<td>YesNo</td>
</tr>
<tr>
<td>RetryCancel</td>
</tr>
</tbody>
</table>

For example, to display OK and Cancel buttons, call

```csharp
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld", MessageBoxButtons.OKCancel);  
```

If you use one of the versions of `MessageBox.Show` without this argument, only the OK button is displayed. The `AbortRetryIgnore` buttons are based on an infamous message that MS-DOS used to display when you tried to access a device (usually a floppy disk) that couldn't respond for some reason. These buttons should probably be avoided in a graphical environment unless you're deliberately trying to be anachronistically humorous.

You can also include one of the values from the `MessageBoxIcon` enumeration to display an icon in the message box:

<table>
<thead>
<tr>
<th><strong>MessageBoxIcon Enumeration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Hand</td>
</tr>
<tr>
<td>Stop</td>
</tr>
<tr>
<td>Error</td>
</tr>
<tr>
<td>Question</td>
</tr>
<tr>
<td>Exclamation</td>
</tr>
<tr>
<td>Warning</td>
</tr>
<tr>
<td>Asterisk</td>
</tr>
<tr>
<td>Information</td>
</tr>
</tbody>
</table>

However, you can see by the values that there are really only four unique message box icons. Here's an example:

```csharp
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld", MessageBoxButtons.OKCancel, MessageBoxIcon.Exclamation);  
```

If you've specified a `MessageBoxButtons` value that displays two or three buttons, you can use the `MessageBoxDefaultButton` enumeration to indicate which button is to be the default:

<table>
<thead>
<tr>
<th><strong>MessageBoxDefaultButton Enumeration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>Button1</td>
</tr>
</tbody>
</table>

```csharp
```
For example, calling

```csharp
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld",
    MessageBoxButtons.OKCancel, MessageBoxIcon.Exclamation,
    MessageBoxDefaultButton.Button2);
```

makes the second button—the button labeled "Cancel"—the default button. That's the button that will be highlighted when the message box first appears and that will respond to keyboard input, such as a press of the space bar.

One other enumeration used by the `Show` method of the `MessageBox` class is `MessageBoxOptions`:

**MessageBoxOptions Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultDesktopOnly</td>
<td>0x20000</td>
</tr>
<tr>
<td>RightAlign</td>
<td>0x080000</td>
</tr>
<tr>
<td>RtlReading</td>
<td>0x100000</td>
</tr>
<tr>
<td>ServiceNotification</td>
<td>0x200000</td>
</tr>
</tbody>
</table>

These options are rarely used, however.

If you're displaying more than one button in the message box, you probably want to know which button the user presses to make the message box go away. That's indicated as the return value from `MessageBox.Show`, which is one of the following enumeration values:

**DialogResult Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>OK</td>
<td>1</td>
</tr>
<tr>
<td>Cancel</td>
<td>2</td>
</tr>
<tr>
<td>Abort</td>
<td>3</td>
</tr>
<tr>
<td>Retry</td>
<td>4</td>
</tr>
<tr>
<td>Ignore</td>
<td>5</td>
</tr>
<tr>
<td>Yes</td>
<td>6</td>
</tr>
<tr>
<td>No</td>
<td>7</td>
</tr>
</tbody>
</table>

Here's how you customarily use the return value from `MessageBox.Show`:

```csharp
DialogResult dr = MessageBox.Show("Do you want to create a new file?", "WonderWord",
    MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question);

if (dr == DialogResult.Yes)
{
    // "Yes" processing
```
else if (dr == DialogResult.No)
{
    // "No" processing
}
else
{
    // "Cancel" processing
}

Or you might want to use a switch and case construction, perhaps like so:

```
switch (MessageBox.Show("Do you want to create a new file?",
    "WonderWord",
    MessageBoxButtons.YesNoCancel,
    MessageBoxIcon.Question)
{
    case DialogResult.Yes:
        // "Yes" processing
        break;

    case DialogResult.No:
        // "No" processing
        break;

    case DialogResult.Cancel:
        // "Cancel" processing
        break;
}
```

Message boxes are sometimes handy for quick exploratory purposes. For example, suppose you want to display the name of the directory that Windows identifies with the alias "My Documents." That information is available from the Environment class in the System namespace. You use the static GetFolderPath method with a single argument—a member of the Environment.SpecialFolder enumeration. The two names separated by a period indicate that SpecialFolder is an enumeration defined within the Environment class.

**MyDocumentsFolder.cs**

```csharp
// MyDocumentsFolder.cs © 2001 by Charles Petzold
using System;
using System.Windows.Forms;

class MyDocumentsFolder
{
    public static void Main()
    {
        MessageBox.Show(
```
The message box looks like this on my system:

![Image of a message box]

**The Form**

Of course, message boxes do not a Windows program make. To begin construction of a full-fledged Windows application, you need to create something that in Windows programming is traditionally called a *window* and in the .NET Framework is called a *form*. A Windows Forms program generally creates a form as its main application window. Applications also use forms for dialog boxes.

A form used as a main application window generally consists of a *caption bar* (sometimes also called a *title bar*) with the name of the application, a *menu bar* underneath that caption bar, and an area inside called the *client area*. A sizing border or (alternatively) a thin border that prevents the form from being resized can surround the whole form. Until **Chapter 14**, however, none of our forms will have menus.

In the pages ahead, we're going to explore several nonstandard and unconventional approaches to creating a form and getting it up on the screen before settling into the most common and approved method. In this way, I hope that you'll get a deeper understanding of what's going on.

Our first effort is what I believe to be the shortest program that actually creates a form. It's called *NewForm.cs*.

**NewForm.cs**

```csharp
using System; // NewForm.cs © 2001 by Charles Petzold
//------------------------------

class NewForm
{
    public static void Main() { new System.Windows.Forms.Form(); }
}
```

The only way this program could be shorter is if I used a shorter class name, got rid of the comments and the extraneous white space, and deleted the `public` access modifier (which isn't strictly needed).

*Form* is a class in the `System.Windows.Forms` namespace. The NewForm program uses the `new` operator to create a new instance of the *Form* class. By now, you know that I could have made the program a bit longer by including a `using` directive,

```csharp
using System.Windows.Forms;
```

at the top of the program, in which case the sole statement in *Main* would be

```csharp
new Form();
```

Or I could have defined an object of type *Form* like so:
Form formOfMine;

and then assigned the result of the `new` operator to that object:
formOfMine = new Form();

Or I could have done both jobs in one line:
Form formOfMine = new Form();

The `Form` class derives from `ContainerControl`, but it actually has a long pedigree beginning with the `Object` class that everything else in the .NET Framework derives from:

![Class inheritance diagram]

The word `control` is used to refer collectively to user interface objects such as buttons, scroll bars, and edit fields; the `Control` class implements much of the base support needed for such objects, in particular, keyboard and mouse input, and visuals. The `ScrollableControl` class adds automatic scrolling support to the control (as we'll explore in Chapter 4), and the `ContainerControl` class allows a control to work like a dialog box as a `parent` to other controls; that is, other controls appear on the surface of the container control.

Although the NewForm program certainly creates a form, it has a bit of a problem. The constructor for the `Form` class stops short of actually displaying the form that it has created. The form is created, but it isn't made visible. As the program terminates, that form is destroyed.

**Showing the Form**

The next version of the program, called ShowForm, corrects that deficiency.

```
using System.Windows.Forms;

class ShowForm
{
    public static void Main()
    {
        Form form = new Form();

        form.Show();
    }
}
```

This version of the program includes a `using` statement that reduces the amount of typing we need to do. Otherwise, both appearances of the uppercase `Form` would have to be prefaced with
System.Windows.Forms. The lowercase form refers to an instance of the Form class created in this
program. You can use whatever name you want. (However, if you're programming in a case-
insensitive language like Visual Basic, you can't use form because the compiler will confuse the
name with the Form class; you'll need to choose a different name for the instance of Form.)

Show is one of two methods that Form inherits from Control that affect the visibility of the form (or the
control):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Show()</td>
<td>Makes a control visible</td>
</tr>
<tr>
<td>void Hide()</td>
<td>Makes a control invisible</td>
</tr>
</tbody>
</table>

An alternative to
form.Show();

is
form.Visible = true;

Show is a method. Visible looks like a field but in fact it's a property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Visible</td>
<td>get/set</td>
</tr>
</tbody>
</table>

ShowForm makes the form visible all right, but you really have to pay attention to see it! Just about
as soon as the form comes up, it disappears on you. If your machine is much faster than mine, you
might not see it at all.

This behavior implies a possible answer to the question I posed about the difference between a
console application and a Windows application: When a command-line program terminates, it leaves
behind its output on the console. When a Windows application terminates, it cleans up after itself by
destroying the window and any output that's displayed.

Could we slow down the program a bit so that we can get a good look at it? Well, are you familiar
with the concept of sleep? If you dig into the System.Threading namespace, you'll find a class
named Thread and a static method of that class named Sleep, which suspends a program (more
accurately, a thread of a program) for a specified period of time in milliseconds.

Here's a program that calls Sleep twice (with arguments indicating 2.5 seconds each) and lets you
get a better look at the form.

ShowFormAndSleep.cs

```
using System.Threading;
using System.Windows.Forms;

class ShowFormAndSleep
{
    public static void Main()
    {
        Form form = new Form();
```
As a bonus, this version of the program also sets the `Text` property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
</tbody>
</table>

*Text* is a very important property. For button controls, the *Text* property indicates the text that the button displays; for edit fields, it's the actual text in the field. For forms, it's the text that appears in the form's caption bar. When you run this program, you first see the form with a blank caption bar for 2.5 seconds; then the caption bar text appears, and 2.5 seconds later, the form goes away.

This is progress of a sort, but I'm afraid that the *Sleep* method isn't the proper way to get a form to stay up on the screen.

**It's an Application and We Want to Run It**

The magic method we need is called *Run*, and it's part of the *Application* class in the *System.Windows.Forms* namespace. Like the *Console* and *MessageBox* classes, the *Application* class can't be instantiated; all its members are defined as static. This program creates a form, sets the form's *Text* and *Visible* properties, and then calls *Application.Run*.

```csharp
using System.Windows.Forms;

class RunFormBadly
{
    public static void Main()
    {
        Form form = new Form();

        form.Text = "Not a Good Idea...";
        form.Visible = true;

        Application.Run();
    }
}
```

Ostensibly, this program is a success. The form it displays looks like this:
You can grab the caption bar with the mouse and move the form around the screen. You can grab the sizing borders and resize the form. You can click the minimize or maximize buttons, you can invoke the system menu (called the control box in Windows Forms) by clicking the icon at the upper left of the window, and you can click the close box in the upper right corner to close the window.

But this program has a very serious flaw that may now become apparent: When you close the form, the Application.Run method never returns and the program remains running even though the form isn't visible. This problem is most obvious if you're compiling the program as a console application: after you close the program, you don't get the familiar "Press any key to continue" text in the Command Prompt window. To terminate the program, you can press Ctrl+C. If you're not compiling the program as a console application, you need to invoke Windows Task Manager, click the Processes tab, find the RunFormBadly application, and manually terminate it. (That's another good reason for compiling as a console application: you can terminate a problem program with Ctrl+C.)

Here's a better way to call Application.Run. You pass the Form object as an argument to the method.

```
//-------------------------------
// RunFormBetter.cs © 2001 by Charles Petzold
//-------------------------------
using System.Windows.Forms;

class RunFormBetter
{
    public static void Main()
    {
        Form form = new Form();

        form.Text = "My Very Own Form";

        Application.Run(form);
    }
}
```

Notice that this version of the program doesn't include a call to Show, and it doesn't set the Visible property either. The form is automatically made visible by the Application.Run method. Moreover,
when you close the form that you've passed to the method, Application.Run returns control back to Main and the program can then properly terminate.

Programmers with experience in the Win32 API might figure out that Application.Run causes the program to enter a message loop and that the form passed to the Run method is equipped with code to post a quit message to the message loop when the form is closed. It is Application.Run that really turns an application into a Windows application.

**Variations on a Theme**

Let's try creating two forms to get a better feel for this process.

**TwoForms.cs**

```csharp
// TwoForms.cs © 2001 by Charles Petzold
using System.Windows.Forms;

class TwoForms
{
    public static void Main()
    {
        Form form1 = new Form();
        Form form2 = new Form();

        form1.Text = "Form passed to Run()";
        form2.Text = "Second form";
        form2.Show();

        Application.Run(form1);

        MessageBox.Show("Application.Run() has returned " +
                         "control back to Main. Bye, bye!",
                         "TwoForms");
    }
}
```

This program creates two forms, named form1 and form2, and gives them two different caption texts so that you can tell them apart. The Show method is called for form2, and form1 is passed to Application.Run. A message box indicates when Application.Run returns control back to Main.

You may want to run TwoForms a couple times to see what happens. If you close form2 first, form1 is unaffected. The only way you can get Application.Run to return and the program to display its message box is to also close form1. If you close form1 first, however, both forms disappear from the screen, Application.Run returns control to Main, and the message box is displayed.

So that's something else that Application.Run does: when you close the form passed as an argument to Application.Run, the method closes all the other forms created by the program. If you don't pass a Form object to Application.Run (as RunFormBadly demonstrated), the program needs to explicitly call the Application.Exit method to force Application.Run to return. But where can the program call Application.Exit if it's off somewhere in the Application.Run call? We'll see shortly how a program can set events that return control to a program and potentially give it the opportunity to call Application.Exit if it needs to.

**Form Properties**
Like many other classes, the Form class defines a number of properties, and Form also inherits additional properties from its ancestors, particularly Control. Two such properties that I've already described are Text and Visible. Here's a program that sets a smattering of sample properties to illustrate some of the flexibility you have in creating and displaying a form.

FormProperties.cs

```csharp
using System.Drawing;
using System.Windows.Forms;

class FormProperties
{
    public static void Main()
    {
        Form form = new Form();

        form.Text = "Form Properties";
        form.BackColor = Color.BlanchedAlmond;
        form.Width *= 2;
        form.Height /= 2;
        form.FormBorderStyle = FormBorderStyle.FixedSingle;
        form.MaximizeBox = false;
        form.Cursor = Cursors.Hand;
        form.StartPosition = FormStartPosition.CenterScreen;

        Application.Run(form);
    }
}
```

BackColor is the property that determines the background color of the form. As you'll see in the next chapter, Color is a structure defined in the System.Drawing namespace (notice the using statement) that contains 141 properties that are actually color names. These names are listed on the inside back cover of this book.

The Width and Height properties determine the initial dimensions of the form. The two statements that change these properties perform both get and set operations, effectively doubling the width of the window and halving its height from the default values.

FormBorderStyle is an enumeration that defines not just the appearance and functionality of the form's border but other aspects of the form as well. Here are the possible values:

**FormBorderStyle Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>No border, no caption bar</td>
</tr>
<tr>
<td>FixedSingle</td>
<td>1</td>
<td>Same as FixedDialog</td>
</tr>
<tr>
<td>Fixed3D</td>
<td>2</td>
<td>Chiseled look</td>
</tr>
<tr>
<td>FixedDialog</td>
<td>3</td>
<td>Preferred for dialog boxes</td>
</tr>
</tbody>
</table>
**FormBorderStyle Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sizable</td>
<td>4</td>
<td>Default</td>
</tr>
<tr>
<td>FixedToolWindow</td>
<td>5</td>
<td>Smaller caption bar, no control box</td>
</tr>
<tr>
<td>SizableToolWindow</td>
<td>6</td>
<td>Same as FixedToolWindow but with sizing border</td>
</tr>
</tbody>
</table>

The default `FormBorderStyle.Sizable` style results in a form that has a caption bar with a control box on the left, followed by the caption bar text; and a minimize box, a maximize box, and a close box at the right. A tool window has a shorter caption bar, no control box, no minimize box, and no maximize box.

The `FormBorderStyle.FixedSingle` style I've used in this program prevents the user from resizing the form. In addition, I've set the `MaximizeBox` property to `false`, so the maximize box is disabled, as shown here:

![Form Properties](image)

The `Cursor` property indicates what the mouse cursor looks like when it's moved to the client area of the form. The `StartPosition` property indicates where the form is initially displayed; the `FormStartPosition` enumeration value `CenterScreen` directs the form to appear in the center of the screen rather than in a default position determined by Windows.

As you look at the FormProperties program, you might start to be puzzled about how Windows Forms programs are structured. It seems like you need to call `Application.Run` to get the form to interact with the user, but `Application.Run` doesn't return until the form is closed.

In short, there doesn't seem to be any place to put your code!

**Event-Driven Input**

Many console programs don't interact with a user at all. A typical console application obtains all the information it needs from command-line arguments, does its stuff, and then terminates. If a console program needs to interact with a user, it gets input from the keyboard. In the .NET Framework, a console program reads keyboard input by calling the `Read` or `ReadLine` methods of the `Console` class. After the program pauses to get keyboard input, it then continues on its way.

Programs written for graphical environments, however, have a different input model. One reason for this is the existence of multiple input devices. Programs get interactive input not only from the keyboard but also from the mouse. In addition, programs can create controls—such as buttons, menus, and scroll bars—that also interact with the user on behalf of the main program.

In theory, I suppose, a programming environment that supported multiple input devices could handle everything using the technique of serial polling. In serial polling, the program checks for input from the keyboard, and if there is none, checks the mouse; if there's none there, it checks for input from the menu, and the menu checks for input from the keyboard and the mouse, and so forth. (Prior to the advent of Windows, character-mode PC programs that used mouse input had to do serial polling.)

It turns out, however, that a better input model for multiple input devices is the event-driven model. As implemented in Windows Forms, each type of input is associated with a different method in a class. When a particular input event occurs (such as a key on the keyboard being pressed, the mouse being moved, or an item being selected from the program’s menu), the appropriate method is called, seemingly from outside the program.

At first, this input model sounds chaotic. As the user is typing away and moving the mouse, pressing buttons, scrolling scroll bars, and picking menu selections, the program must get bombarded with
method calls coming from all different directions. Yet in practice, it's much more orderly than it sounds because all the methods exist in the same execution thread. Events never interrupt a program's execution. Only when one method finishes processing its event is another method called with another event.

Indeed, after a Windows Forms program performs initialization on its form, every little piece of code it executes—is in response to an event. For much of the time, the program is sitting dormant, somewhere deep inside the Application.Run call, waiting for an event to happen. Indeed, it's often helpful to think of your Windows Forms programs as state machines whose state is determined entirely by changes initiated by events.

Events are so important that they are woven into the very fabric of the .NET Framework and C#. Events are members of classes along with constructors, fields, methods, and properties. When a program defines a method to process an event, the method is called an event handler. The arguments of the handler match a function prototype definition called a delegate. We'll see how this all works shortly.

As you'll discover in Chapter 6, there are three different types of keyboard events. One type of event tells you when a key is pressed and another when the key is released. A third keyboard event tells you when a character code has been generated by a particular combination of keystrokes.

In Chapter 8, I'll introduce the seven types of mouse events, indicating when the mouse has moved and what buttons have been clicked or double-clicked.

In Chapter 10, you'll see that there's also a timer event. This event periodically notifies your form when a preset length of time has elapsed. Clock programs use timer events to update the time every second.

In Chapter 12, when we start creating controls (such as buttons and text boxes and list boxes) and putting them on the surface of forms, you’ll find out that these controls communicate information back to the form with events. Events indicate when the button has been clicked or the text in the text box has changed.

In Chapter 14, you'll discover that menus also communicate information to a form using events. There's an event to indicate when a drop-down menu is about to be displayed, an event to indicate when a menu item is selected, and an event to indicate when a menu item is clicked.

But one of the oddest events—perhaps the most unlikely candidate for eventhood—is also one of the most important. This event, known as the Paint event, tells your program when you need to display output on your window.

Nothing reveals the enormous difference between command-line programs and graphical programs more than the Paint event. A command-line program displays output whenever it feels like it. A Windows Forms program can display output whenever it wants to as well, but doing so isn't quite adequate. What the Paint event is really doing is informing a program when part or all of the form's client area is invalid and must be redrawn.

How does a client area become invalid? When a form is first created, the entire client area is invalid because the program hasn't yet drawn anything. The first Paint event that a program receives tells it to draw something on the client area.

When you move windows around the screen so that they overlap, Windows doesn't save the appearance of a client area that is covered by another window. When that client area is later uncovered, the program must restore its appearance. For that reason, it gets another Paint event. When you restore a program that's been minimized, you get another Paint event.

A Windows program must be able to entirely repaint its client area at any time. It must retain—or keep quickly accessible—all the information it needs to do this. Structuring your programs to respond properly to Paint events may sound quite restrictive, but you’ll get the hang of it.

Handling the Paint Event

The subject of events is best approached with examples. In practical terms, handling a Paint event in your program first involves taking a look at PaintEventHandler. PaintEventHandler is a delegate that
is defined in the `System.Windows.Forms` namespace with a single statement that (in C# syntax) looks like this:

```csharp
public delegate void PaintEventHandler(object objSender, PaintEventArgs pea);
```

If this statement looks like a function prototype to you, you’re not too far from the mark. The second argument indicates a class named `PaintEventArgs`—also defined in the `System.Windows.Forms` namespace—that I’ll discuss shortly.

To handle `Paint` events in one of the programs shown earlier in this chapter, you must define a static method in your class that has the same arguments and return type as the `PaintEventHandler` delegate:

```csharp
static void MyPaintHandler(object objSender, PaintEventArgs pea) {
}
```

You then attach this event handler to the `Paint` event of the `Form` class with some very special syntax that looks like this:

```csharp
form.Paint += new PaintEventHandler(MyPaintHandler);
```

`Paint` is an event defined in the `Control` class and is part of the `Form` class by virtue of inheritance. The only two operations you can perform on the `Paint` event are the assignment operators `+=` and `−=`. The `+=` operator installs an event handler by attaching a method to an event. The general syntax is

```csharp
object.event += new delegate(method)
```

You detach a method from an event by using the same general syntax but with the `−=` operator:

```csharp
object.event -= new delegate(method)
```

Detaching a method from an event is rarely necessary, however. Generally, you’ll install an event handler and never uninstall it.

The two arguments to the `Paint` event handler are an object I’ve called `objSender` and a `PaintEventArgs` class I’ve abbreviated as `pea`. The first argument refers to the object that this `Paint` event applies to, in this case, the object `form`. The object is called a "sender" because the event originates from that object.

The `PaintEventArgs` class is defined in the `System.Windows.Forms` namespace, and it has two properties, named `Graphics` and `ClipRectangle`, which are both read-only:

<table>
<thead>
<tr>
<th><code>PaintEventArgs</code> Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Graphics</td>
</tr>
<tr>
<td>Rectangle</td>
</tr>
</tbody>
</table>

The `Graphics` property contains an instantiation of the `Graphics` class, which is defined in the `System.Drawing` namespace. `Graphics` is an extremely important class in the Windows Forms library, ranking right up there with `Form`. This is the class you use to draw graphics and text on your form. The `System.Drawing` namespace implements a graphics programming system known as GDI+, which is an enhanced version of the Windows Graphics Device Interface. I’ll discuss the `ClipRectangle` property in [Chapter 4](#).

In a vast majority of the programs in this book, you’ll see

```csharp
Graphics grfx = pea.Graphics;
```
as the first line in the Paint event handler. You can name your Graphics object whatever you want. Some programmers use the lowercase graphics, but this object shows up so much in graphics code that some programmers use just the letter g! I've taken a compromise approach.

Before all this new stuff piles up too deeply, let's take a look at an actual program that implements a Paint event handler.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class PaintEvent
{
    public static void Main()
    {
        Form form = new Form();
        form.Text = "Paint Event";
        form.Paint += new PaintEventHandler(MyPaintHandler);

        Application.Run(form);
    }

    static void MyPaintHandler(object objSender, PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        grfx.Clear(Color.Chocolate);
    }
}
```

After the form is created in Main, the method named MyPaintHandler is attached to the Paint event of the form. In this handler, the program obtains a Graphics object from the PaintEventArgs class and uses that to call the method Clear. Clear is a simple method—perhaps the simplest drawing method—defined in the Graphics class:

### Graphics Methods (selection)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Clear(Color clr)</td>
<td>Paints entire client area with color</td>
</tr>
</tbody>
</table>

The argument is an object of type Color, which I'll discuss in much more detail in the next chapter. As I mentioned in connection with the FormProperties program shown earlier in this chapter, the easiest way to get a color is to specify one of the 141 color names implemented as static properties in the Color structure.

To get an idea of the frequency with which the program gets Paint events, try inserting the statement `Console.WriteLine("Paint Event");` in MyPaintHandler. A couple programs in the next chapter will also visually demonstrate the frequency of Paint events.
From here on, all the Windows Forms programs in this book will have at least the following three
using statements at the top of the program:

using System;
using System.Drawing;
using System.Windows.Forms;

Generally, these are the minimum required for any nontrivial Windows Forms application.

You might see a connection between these three using statements and the three DLLs that you
need to specify as references when compiling the program. It's natural for a C or C++ programmer to
think of the using statements as equivalent to #include statements. They are not! They're a little
more closely related to the With statement in Visual Basic. They exist solely so that you don't have to
type fully qualified class names. Everything that header files normally provide in C and C++
programs (such as type declarations, function declarations, and class declarations) is provided
instead by the DLLs specified as references, the same DLLs that are linked with the running program
to implement these classes.

Displaying Text

The Graphics class has many methods to draw graphics figures such as lines, curves, rectangles,
elipses, and bitmapped images. The Graphics method that displays text in a form is called
DrawString (not to be confused with the cord that may be holding up your pants).

DrawString comes in six overloaded versions, but the first three arguments are always the same. At
this point in our lives, the simplest version of DrawString is defined like so:
void DrawString(string str, Font font, Brush brush, float x, float y)

You might expect the arguments of DrawString to include the text string you want to display and the
coordinate position where it is to appear. You might not expect the method to also include the font
used to display the text and something called a Brush (which is used to color the text), but there they
are. The presence of these two arguments is part of what is implied when GDI+ is said to be a
stateless graphics programming system. Just about everything that the system needs to display
various graphics figures is included right in the method calls.

The downside is that the DrawString call is rather bulky with information. You might find yourself
reducing the second and third arguments to single letters or searching out other ways to make the
method call less lengthy.

The first argument to DrawString is the text string you want to display, for example,
grfx.DrawString("Hello, world!", ...);

Let's take a look at the other arguments in detail.

The Font

The second argument to DrawString is the font used for drawing the text. This is an object of type
Font, a class defined in the System.Drawing namespace. I'll have much more to say about the Font
class in Chapter 9. Suffice it to say that a Windows Forms program has access to many fonts with
scalable sizes. For now, we'll use a default font. Very conveniently, every class derived from Control
inherits a property named Font that stores the default font for the control.

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font</td>
<td>Font</td>
<td>get/set</td>
<td>Default font for the control</td>
</tr>
</tbody>
</table>

You might find it quite confusing at first to deal with a class and a property that are both named Font,
but I assure you, after some months, you'll find it somewhat less confusing.

When you install a Paint event handler for a form, you can obtain the object that the event applies to
by casting the first argument to the type of that object:
Form form = (Form)objSender;

This cast works because objSender is indeed an object of type Form. If objSender were not an object of type Form (or a class descended from Form), this statement would raise an exception. Thus, within the event handler, you can reference the default font for the form by using form.Font. The DrawString call thus looks something like this:

grfx.DrawString(str, form.Font, ...);

If you have multiple DrawString calls, you might first want to define an object of type Font and assign the form's default font to it:

Font font = form.Font;

That statement includes a lot of font! The first Font is the class defined in the System.Drawing namespace. The lowercase font is an object of that class. The last Font is a property of the Form class. The DrawString call then becomes

grfx.DrawString(str, font, ...);

To be more concise, you could name this Font object just f.

The Brush

The third argument to DrawString indicates the "color" of the font characters. I put "color" in quotation marks because the argument is actually an object of type Brush, and brushes can be much more than just color. Brushes can be gradients of color or fancy patterns or bitmapped images. Indeed, brushes are so wonderfully varied and powerful that they get very nearly their own entire chapter. But since that is Chapter 17 and this is Chapter 2, for now we'll have to be satisfied with very simple brushes.

The simplest way to be colorfully versatile is with the Brushes class. Notice the plural Brushes and not the singular Brush, which is also the name of a class. The Brushes class has 141 static read-only properties with the same color names as implemented in the Color class and listed on the inside back cover of this book. The Brushes properties return objects of type Brush. Because these are static properties, they are referenced using the class name and property name, like the example here:

grfx.DrawString(str, font, Brushes.PapayaWhip, ...);

You're probably thinking, "Sure it might be fun drawing text with lots of different colors and maybe gradients and patterns and stuff like that, but let's be realistic: Probably 97.5 percent of the text I'll want to display will be plain old black. With few exceptions, I'll probably just use Brushes.Black as the third argument to DrawString." So, you can define an object of type Brush like so:

Brush brush = Brushes.Black;

and pass that object to 97.5 percent of your DrawString calls:

grfx.DrawString(str, font, brush, ...);

You could, of course, even name it b to do less typing.

But I'm afraid that using Brushes.Black in this way would be a mistake. You're making an implicit assumption that the background of the form isn't also black. Could it be? Yes, and very easily. In such a case, the text wouldn't be visible.

Regardless, for now I'll give you special dispensation to use Brushes.Black in calls to DrawString, but only if you also set the BackColor property of the form to Color.White or something else that's guaranteed to make the black text visible. I'll discuss better approaches to selecting colors in Chapter 3.

The Coordinate Points

Finally, the last two arguments of DrawString indicate the horizontal (x) and vertical (y) coordinates where the upper left corner of the text string is to appear.
If you come from a mathematics background—or if the trauma of high school mathematics has forever left its scar on your brain—you may have envisioned a two-dimensional coordinate system like so:

![Cartesian coordinate system diagram]

This is known as a *Cartesian coordinate system*, after French mathematician and philosopher René Descartes (1596–1650), who is credited with inventing analytical geometry, and to whom the field of computer graphics is eternally indebted. The origin—the point (0, 0)—is in the center. Values of $x$ increase to the right, and values of $y$ increase going up.

However, this isn't exactly the coordinate system used in most graphical environments. A coordinate system in which increasing values of $y$ go up is at odds with the way in which most Western languages are written. Also, early computer graphics involved programmers writing directly into video display memory. Video memory buffers are arranged starting at the top of the screen because computer monitors scan from the top down. And that's because television sets scan from the top down, and that decision goes back some 60 years or so.

In the Windows Forms environment, as in most graphical environments, the default coordinate system has an origin in the upper left corner and looks like this:

![Default coordinate system diagram]

I say this is the *default* coordinate system because it's possible to change it to something else. Such fun awaits us in Chapter 7.

When you draw on a form using the `Graphics` object that you obtain from the `PaintEventArgs` class passed as an argument to your `Paint` event handler, all coordinates are relative to the upper left corner of the client area of the form. All units are in pixels. Increasing values of $x$ go to the right, and increasing values of $y$ go down.

Let me repeat: Coordinates are relative to the upper left corner of the *client area*. The client area is the area inside a form that's not occupied by the form's caption bar or sizing border or any menu the form might have. When you use the `Graphics` object from the `PaintEventArgs` class, you can't draw outside the client area. This means you never have to worry about drawing something where you're not supposed to.

The coordinate point passed to the `DrawString` method refers to the position of the upper left corner of the first character of the text string. If you specify a coordinate of $(0, 0)$, the text string is thus displayed in the upper left corner of the client area.

So let's put it all together in a program called `PaintHello`.

*PaintHello.cs*
using System;
using System.Drawing;
using System.Windows.Forms;
class PaintHello
{
    public static void Main()
    {
        Form form = new Form();
        form.Text = "Paint Hello";
        form.BackColor = Color.White;
        form.Paint += new PaintEventHandler(MyPaintHandler);

        Application.Run(form);
    }
    static void MyPaintHandler(object objSender, PaintEventArgs pea)
    {
        Form form = (Form)objSender;
        Graphics grfx = pea.Graphics;

        grfx.DrawString("Hello, world!", form.Font, Brushes.Black, 0, 0);
    }
}

And here we have our first—but, as you'll see, perhaps not quite the simplest—program that displays text in a form. The text appears in the upper left corner of the client area:

![Paint Hello](image)

Hello, world!

---

An exception is the OS/2 Presentation Manager, which was designed as a completely bottom-up system. This was fine for graphics programming but didn't always work otherwise. Programmers had to use bottom-up coordinates when specifying the location of controls in dialog boxes, for example, which often entailed designing the dialog box from the bottom up. See Charles Petzold, *Programming the OS/2 Presentation Manager* (Redmond, WA: Microsoft Press, 1989) or Charles Petzold, *OS/2 Presentation Manager Programming* (Emeryville, CA: Ziff-Davis Press, 1994) for details.

**The Paint Event Is Special!**

Watch out what you put in the *Paint* event handler. The method can be called quite frequently and sometimes unexpectedly, and it works best when it can repaint the client area quickly without interruption.

Earlier in this chapter, I suggested that you use message boxes for simple debugging. But don't put a call to `MessageBox.Show` in the *Paint* event handler! The message box could cover up part of the client area and result in another *Paint* event. And another and another and another.... Also, don't put any `Console.Read` or `Console.ReadLine` calls in there or in any event handler. `Console.Write` or `Console.WriteLine` calls are safe, however.

And don't do anything that accumulates. In one of my very early Windows Forms programs, I wrote a *Paint* event handler that accessed the *Font* property, made a new font that was twice as big, and set the *Font* property to that new font. Well, every time there was a new *Paint* event, the font got twice as big as the time before. It was like *Honey, I Blew Up the Font*.

Doing all your drawing in the *Paint* event handler might sound a bit restrictive, and at times it is. That's why Windows Forms implements a couple methods to make painting more flexible.

First, you can obtain a *Graphics* object outside a *Paint* event handler by calling the `CreateGraphics` method implemented in *Control* and inherited by *Form*. Second, at times, you'll need to generate a *Paint* event from some other event. The method that does this is `Invalidate`, which is implemented in the *Control* class. I'll demonstrate how to do these things when covering keyboard, mouse, and timer input in Chapters 6, 8, and 10.

**Multiple Forms, Multiple Handlers**

To get a better feel for the *Paint* event handler, let's look at a couple variations on the basic theme. This program uses the same *Paint* event handler for two forms that it creates.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class PaintTwoForms
{
    static Form form1, form2;

    public static void Main()
    {
        form1 = new Form();
        form2 = new Form();

        form1.Text = "First Form";
    }
}
```
form1.BackColor = Color.White;
form1.Paint    += new PaintEventHandler(MyPaintHandler);

form2.Text      = "Second Form";
form2.BackColor = Color.White;
form2.Paint    += new PaintEventHandler(MyPaintHandler);
form2.Show();

Application.Run(form1);
}

static void MyPaintHandler(object objSender, PaintEventArgs pea)
{
Form     form = (Form)objSender;
Graphics grfx = pea.Graphics;
string   str;

if(form == form1)
    str = "Hello from the first form";
else
    str = "Hello from the second form";

    grfx.DrawString(str, form.Font, Brushes.Black, 0, 0);
}

Notice that the Form objects are stored as fields so that they are accessible from both Main and the Paint event handler. Each call to the Paint event handler applies to one of the two forms the program created. The event handler can determine which form it applies to by comparing the objSender argument (cast to a Form object) with the two Form objects stored as fields. If you don't mind a little capitalization problem, you could replace the entire if and else construction with the single statement

str = "Hello from the " + form.Text;

Now let's try just the opposite. Let's create one form but attach two Paint event handlers to it.

TwoPaintHandlers.cs
{//-------------------------------
// TwoPaintHandlers.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoPaintHandlers
{
    public static void Main()
    {
        Form form      = new Form();
        form.Text      = "Two Paint Handlers";
        form.BackColor = Color.White;
        
        form1.BackColor = Color.White;
        form1.Paint    += new PaintEventHandler(MyPaintHandler);

        form2.Text      = "Second Form";
        form2.BackColor = Color.White;
        form2.Paint    += new PaintEventHandler(MyPaintHandler);
        form2.Show();

        Application.Run(form1);
    }
}

TwoPaintHandlers.cs
{//-------------------------------
// TwoPaintHandlers.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoPaintHandlers
{
    public static void Main()
    {
        Form      form = new Form();
        form.Text   = "Two Paint Handlers";
        form.BackColor = Color.White;

        form1.BackColor = Color.White;
        form1.Paint    += new PaintEventHandler(MyPaintHandler);

        form2.Text      = "Second Form";
        form2.BackColor = Color.White;
        form2.Paint    += new PaintEventHandler(MyPaintHandler);
        form2.Show();

        Application.Run(form1);
    }
}
form.Paint += new PaintEventHandler(PaintHandler1);
form.Paint += new PaintEventHandler(PaintHandler2);

Application.Run(form);
}
static void PaintHandler1(object objSender, PaintEventArgs pea)
{
    Form form = (Form)objSender;
    Graphics grfx = pea.Graphics;
    
    grfx.DrawString("First Paint Event Handler", form.Font, 
                   Brushes.Black, 0, 0);
}
static void PaintHandler2(object objSender, PaintEventArgs pea)
{
    Form form = (Form)objSender;
    Graphics grfx = pea.Graphics;
    
    grfx.DrawString("Second Paint Event Handler", form.Font, 
                   Brushes.Black, 0, 100);
}

This program highlights one of the interesting aspects of attaching handlers to events. If there is more than one handler, all the handlers get called in sequence. Notice that the `DrawString` coordinates are (0, 0) in the first handler and (0, 100) in the second handler. I'm making an assumption that the default font isn't more than 100 pixels tall, but that seems fairly safe.

Inheriting Forms

So far, you've seen how you can create a form, give it some properties (such as a text string to show in its caption bar and a nondefault background color), and attach some event handlers. Just as you attached a `Paint` event handler, you can attach handlers for the keyboard, mouse, menus, and so forth.
But I'm afraid the truth is this: it's not usually done like that.

To exploit the full power of everything implemented in the Form class, you can't just create a form. You must become a form. For just as Control begat ScrollableControl, and ScrollableControl begat ContainerControl, and ContainerControl begat Form, then Form can now beget some truly amazing form that only you can create.

You create such a form in your program by defining a class that inherits from Form. Let's take a look.

InheritTheForm.cs

```
//---------------------------------------------
// InheritTheForm.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class InheritTheForm: Form
{
    public static void Main()
    {
        InheritTheForm form = new InheritTheForm();
        form.Text = "Inherit the Form";
        form.BackColor = Color.White;

        Application.Run(form);
    }
}
```

Let me draw your attention to the class statement:

```
class InheritTheForm: Form
```

The part of the statement that follows the class name, : Form, means that InheritTheForm is a descendent of Form and inherits every method and property of Form.

This class still has a static Main method that is the entry point to the program. However, Main creates a new instance of InheritTheForm rather than Form. Because InheritTheForm derives from Form, of course it also has properties named Text and BackColor, which the program sets next. Just as an object of type Form can be passed to Application.Run, any object of a type derived from Form can also be passed to Application.Run.

The InheritTheForm program creates the form, performs initialization (which in this case just involves setting the Text property), and then passes the form object to Application.Run. A more conventional approach is to move form initialization to the class's constructor.

InheritWithConstructor.cs

```
//---------------------------------------------
// InheritWithConstructor.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class InheritWithConstructor: Form
```
{ 
    public static void Main()
    {
        Application.Run(new InheritWithConstructor());
    }
    public InheritWithConstructor()
    {
        Text = "Inherit with Constructor";
        BackColor = Color.White;
    }
}

You'll recall that a constructor has no return type, and a default constructor has an empty argument list.

Form has a pedigree starting at Object and encompassing five other classes. When an InheritWithConstructor object is created in Main, first the default constructor for Object is called, then the default constructor for the MarshalByRefObject class, and so forth on through the default constructor for the Form class, and finally the default constructor for the InheritWithConstructor class.

Notice that I don't have to preface the Text and BackColor properties with an object name, an object that I called form in previous programs in this chapter. These properties don't need anything in front of them because they are properties of the InheritWithConstructor class. They are properties of InheritWithConstructor because this class derives from Control and Form, in which these properties and many others were originally defined.

If I wanted to preface these properties with anything, it would be the keyword this:

```
this.Text = "Inherit with Constructor";
this.BackColor = Color.White;
```

The this keyword indicates the current object.

The OnPaint Method

What advantages do you get by inheriting Form rather than just creating an instance of it? Although most of the methods and properties implemented in Form are defined as public, some essential ones are defined as protected. These protected methods and properties can be accessed only by a descendent of Form. One such protected property is ResizeRedraw, which I'll be discussing in Chapter 3.

One protected method inherited by Form by way of Control is named OnPaint. You don't want to call this method, however; you want to override it, for if you do, you don't have to install a Paint event handler. The OnPaint method has a single argument, which is an object of type PaintEventArgs. You can use this argument to obtain a Graphics object just as in a Paint event handler.

And here's my final version of a Windows Forms hello-world program.

```csharp
HelloWorld.cs
//-----------------------------------------
// HelloWorld.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloWorld: Form
{
```
public static void Main()
{
    Application.Run(new HelloWorld());
}

public HelloWorld()
{
    Text = "Hello World";
    BackColor = Color.White;
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    grfx.DrawString("Hello, Windows Forms!", Font,
                    Brushes.Black, 0, 0);
}

This is the official, certified, programmer-tested and mother-approved way to create a form in C# using the Windows Forms class library. That's why this is the first program in this book to be called simply HelloWorld. (In the next chapter, I'll show you a better way to specify the background and text colors, however.) Again, notice in OnPaint that I don't have to preface Font with anything. The OnPaint method doesn't need an objSender argument because the form that the OnPaint call applies to is always this.

And here's what it looks like:

![Hello World form]

Of course, there's always some smart aleck in the back row with a raised hand and the impudent question, "Can you now center that text in the window?"

Yes, and in the next chapter, I'll show you three different ways to do it.

**Does Main Belong Here?**

When you look at a program like HelloWorld, you may find yourself wondering about Main. Main is a method in the HelloWorld class, yet Main also creates an instance of the HelloWorld class. This may
seem odd. It may appear as if the program is pulling itself up by its bootstraps. How can Main execute at all when an instance of the HelloWorld class hasn't been created yet?

The answer is that Main is defined as static. Static methods exist independently of any objects that are instantiated from the class. Conceptually, the operating system loads the program into memory and begins execution by making a call to

HelloWorld.Main();

It couldn’t make this call unless Main were defined as static, and if you remove static from the definition of Main, the compiler will complain that the program doesn’t have an entry point.

Still, however, you may be more comfortable putting Main in a class by itself, like the C# programs in Chapter 1. There’s nothing wrong with that approach, and some programmers prefer it. This sample program is named SeparateMain.cs and is functionally equivalent to the HelloWorld program.

SeparateMain.cs

 //---------------------------------------------
 // SeparateMain.cs © 2001 by Charles Petzold
 //---------------------------------------------
 using System;
 using System.Drawing;
 using System.Windows.Forms;

class SeparateMain
{
    public static void Main()
    {
        Application.Run(new AnotherHelloWorld());
    }
}
class AnotherHelloWorld: Form
{
    public AnotherHelloWorld()
    {
        Text = "Another Hello World";
        BackColor = Color.White;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        grfx.DrawString("Hello, Windows Forms!", Font,
                        Brushes.Black, 0, 0);
    }
}

I must admit that this program looks architecturally cleaner to me, and I briefly toyed with structuring all the sample programs in this book like this. However, adding three lines to every program wasn’t appealing to me, and coming up with twice as many class names didn’t make sense either. I was also dissuaded when I began taking notice of all the static methods and properties in the .NET classes that return instances of the class they belong to. Chocolate, for example, is a static property of the Color class but returns an instance of Color.
Events and "On" Methods

As you've seen, when you create an instance of Control or any class derived from Control (such as Form), you can install a Paint event handler by defining a static method with the same return types and arguments as the PaintEventHandler delegate:

```csharp
static void MyPaintHandler(object objSender, PaintEventArgs pea)
{
    // Painting code
}
```

You then install this paint handler for a particular object (named form, for example) using the code

```csharp
form.Paint += new PaintEventHandler(MyPaintHandler);
```

In a class derived from Control, however, you don't need to install a Paint event handler (even though you can). You can simply override the protected OnPaint method:

```csharp
protected override void OnPaint(PaintEventArgs pea)
{
    // Painting code
}
```

You'll find that all events defined in Windows Forms are similar. Every event has a corresponding protected method. The method has a name that consists of the word On followed by the event name. For each event that we'll encounter, I'll show a little table like this:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paint</td>
<td>OnPaint</td>
<td>PaintEventHandler</td>
<td>PaintEventArgs</td>
</tr>
</tbody>
</table>

The table indicates the name of the event, the corresponding method, the delegate involved in installing an event handler, and the argument to the event handler and the method.

You might assume—as I did originally—that the OnPaint method is basically just a preinstalled Paint event handler. But that's wrong. It's really implemented the other way around: the OnPaint method in Control is actually responsible for calling all the installed Paint handlers.

Let's explore this concept a bit. First, just as the HelloWorld class shown earlier inherited from Form, here's a class named InheritHelloWorld that inherits from HelloWorld.

InheritHelloWorld.cs

```csharp
//------------------------------------------------
// InheritHelloWorld.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class InheritHelloWorld: HelloWorld
{
    public new static void Main()
    {
        Application.Run(new InheritHelloWorld());
    }
    public InheritHelloWorld()
```
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    grfx.DrawString("Hello from InheritHelloWorld!",
                    Font, Brushes.Black, 0, 100);
}

Let me take care of some housekeeping issues first. When I created the InheritHelloWorld project in Visual Studio .NET, I created a new C# file named InheritHelloWorld.cs, as usual, but I also needed to include HelloWorld.cs in the project. I did that by using the Add Existing Item option and specifying Link File in the drop-down menu next to the Open button. That avoids making a second copy of the HelloWorld.cs file.

Notice that the Main method includes the new keyword, indicating that it is supposed to replace any Main methods that may be in any parent classes (such as HelloWorld). You also have to tell Visual Studio .NET which Main you want to be the entry point to the program. You do this with the project's Properties dialog box. In the General Common Properties, specify the Startup Object as InheritHelloWorld.

If you're running the command-line C# compiler, specify both source code files in the command line and use the compiler switch
/Main:InheritHelloWorld

to indicate which class has the Main method you want as the entry point to the program.

As I mentioned earlier, when you create a new object based on a derived class using a default constructor, all the ancestral default constructors are called starting with Object. Toward the end of this process, the HelloWorld constructor gets called and responds by setting the Text property of the form to "Hello World." Finally, the InheritHelloWorld constructor is executed and sets the Text property like so:

        Text = "Inherit " + Text;

That the caption bar of this program reads "Inherit Hello World" demonstrates that this sequence of events is correct.

The OnPaint method in InheritHelloWorld overrides the OnPaint method in HelloWorld. When InheritHelloWorld runs, it displays "Hello from InheritHelloWorld!" I've positioned the text at the coordinate position (0, 100) so you can see that the OnPaint method in HelloWorld isn't also executed. The OnPaint method in HelloWorld is overridden.

Now let's take a look at a program that does something a little different. This program doesn't define a class that inherits HelloWorld; this one instantiates the HelloWorld class.

InstantiateHelloWorld.cs
//----------------------------------------------------
// InstantiateHelloWorld.cs © 2001 by Charles Petzold
//----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class InstantiateHelloWorld
{
    public static void Main()
    {
        Form form   = new HelloWorld();
        form.Text   = "Instantiate " + form.Text;
        form.Paint += new PaintEventHandler(MyPaintHandler);

        Application.Run(form);
    }
    static void MyPaintHandler(object objSender, PaintEventArgs pea)
    {
        Form     form = (Form)objSender;
        Graphics grfx = pea.Graphics;

        grfx.DrawString("Hello from InstantiateHelloWorld!",
                       form.Font, Brushes.Black, 0, 100);
    }
}

Take a close look at this code. First, notice that the InstantiateHelloWorld class doesn't inherit from HelloWorld or Form or anything else (except the Object class, of course):

class InstantiateHelloWorld

Instead, it creates a new instance of the HelloWorld class and saves it in the variable form, just as early programs in this chapter created instances of the Form class:

Form form = new HelloWorld();

This program can save the HelloWorld object in a variable of type Form because HelloWorld is derived from Form. During the creation of the HelloWorld object, the HelloWorld constructor is called, which gives the form a Text property of "Hello World." The next statement prepends the word Instantiate to the Text property. The program then installs a Paint event handler for the form.

But what appears in InstantiateHelloWorld's client area is not the text "Hello from InstantiateHelloWorld!" but instead the text "Hello, Windows Forms!" which is what the OnPaint method in HelloWorld displays. What happened?

The OnPaint method in Control is responsible for calling the installed Paint event handlers. Because the HelloWorld class overrides OnPaint, that job doesn't get done. That's why the .NET documentation recommends that when you override one of the protected methods beginning with the On prefix, you should call the On method in the base class like so:

base.OnPaint(pea)

Try inserting this statement at the top of HelloWorld's OnPaint method and rebuilding InstantiateHelloWorld. Now the program works as you probably wanted it to. InstantiateHelloWorld displays its text string ("Hello from InstantiateHelloWorld!") and also the "Hello, Windows Forms!" text string.

The sequence of events in the revised version is this:

- Whenever the client area becomes invalid, the OnPaint method is called. This is the OnPaint method in the HelloWorld class, which overrides any OnPaint method in ancestral classes.
- The OnPaint method in HelloWorld calls the OnPaint method in its base class. (Remember, I'm talking about a revised version of HelloWorld that includes the base.OnPaint call.) That would normally be the OnPaint method implemented in Form, but it's likely Form doesn't override the OnPaint method and what really gets called is the OnPaint method back in Control.
The `OnPaint` method in `Control` calls all the installed `Paint` event handlers. The only one in this process is the `MyPaintHandler` method in `InstantiateHelloWorld`. That method displays some text at position (0, 100).

When all the installed `Paint` event handlers have been called, the `OnPaint` method in `Control` returns back to the `OnPaint` method in `HelloWorld`.

The `OnPaint` method in `HelloWorld` displays some text at position (0, 0).

The Windows Forms documentation recommends that whenever you override an `On` method you call the base class `On` method. However, in most cases, you need to do this only if you're defining a class that you'll also be instantiating, and that the instantiated classes are also installing event handlers for `On` methods you've overridden. This scenario doesn't happen very often. Still, at times, you need to call the base class in overrides of `On` methods. As we'll see in the next chapter, one of these is `OnResize`. 
Chapter 3: Essential Structures

Overview

Computers were originally built to perform numeric calculations, and crunching numbers is still what computers do best. Virtually all programming languages have mechanisms for storing numbers in variables, performing arithmetical operations, looping through ranges of numbers, comparing numbers, and displaying numbers in a readable form.

For many programming languages, the next step beyond numbers is text in the form of character strings. Character strings chiefly exist to allow computer programs to communicate with human users. Internal to the computer, of course, characters are represented by numbers, as is everything else in the machine.

It is the central premise of object-oriented programming that data types beyond the standard numbers and character strings be easy to define and easy to work with. We've already seen several examples of that ease of use, including a class named Form, which hardly seems like either a number or a character string.

In programming for graphical environments, four other data types appear quite frequently:
- Two-dimensional coordinate points
- Two-dimensional sizes in terms of width and height
- Rectangles
- Colors

These four data types are the focus of this chapter.

Classes and Structures

These four data types—actually seven because three of them are implemented in both integer and floating-point forms—are defined in the System.Drawing namespace. Interestingly enough, these seven data types are not implemented as classes. They are instead structures, and indeed, these are seven of only eight structures defined in System.Drawing.

The structure data type (which is defined using the keyword struct) comes to C# by way of C and C++. (Java doesn't have a structure data type.) In C++, classes and structures are very similar. In C++ structures, all methods and fields are public by default, and in C++ classes, all methods and fields are private by default. Of course, you can use the public, private, or protected keywords to change the visibility of any method or field in a C++ class or structure.

In C#, classes and structures are also very similar, but the differences aren't the same as those in C++. In C#, all methods and fields (as well as properties and events) are private by default in both classes and structures. The difference lies in the fact that classes are reference types, and structures are value types. Let's examine what this difference means. Consider the following expression:

```
new Form()
```

This statement causes a memory block to be allocated in an area of general-purpose memory known as the heap. This memory block must be large enough for an instance of the Form object, which means that it must be for large enough for all of Form's instance (that is, nonstatic) fields. If you look at the .NET documentation, that amount of space might not seem like much, but remember that you're not seeing the private fields, and there are undoubtedly many of them.

The value returned from that new expression is essentially a pointer to the memory block located in the heap. That memory pointer is what's saved in the variable form in a statement like this:

```
Form form = new Form();
```

That's what a reference type means: the object is a pointer to (references) a memory block.

Suppose you do something like this:

```
Form form2 = form;
form2.Text = "Form 2 Text";
```
Go ahead: insert these calls into the PaintEvent program from Chapter 2, right before the call to `Application.Run`. What happens? The form that we're displaying—the form referenced by `form`—gets the caption bar text "Form 2 Text." How can this be? It's because `form` is a pointer. The statement

```csharp
Form form2 = form;
```

simply copies that pointer to `form2`. The statement does not create a new instance of the `Form` class. The variables `form` and `form2` are equal, which means they point to the same memory block and therefore refer to the same object.

Now obviously, passing pointers around isn't something you want happening universally. Consider the following sequence of statements:

```csharp
int a = 5;
int b = a;
a = 10;
```

You wouldn't want this to mean that `a` and `b` were identical pointers that would always refer to the same number, and that `b` was now equal to 10! That would be insane. And that's why numbers in C# are value types. The variable name of any value type is not a pointer to a location in memory that stores the number. The variable name represents the number itself.

If you check through the documentation of the `System` namespace, you'll find that most of the basic types—`Boolean`, `Byte`, `Char`, `Decimal`, `Double`, `Int16`, `Int32`, `Int64`, `SByte`, `Single`, `UInt16`, `UInt32`, `UInt64`—are defined as structures rather than classes. Structures inherit from `ValueType`, which inherits from `Object`. You can think of value types as "lightweight objects," and indeed, you should use `struct` only for types that are small and that might be frequently created and destroyed.

### Two-Dimensional Coordinate Points

One data type that is prevalent enough in graphical environments and small enough to justify making it a structure rather than a class is a coordinate point, represented in the .NET Framework by the structure `Point`. In a two-dimensional coordinate system (such as the surface of a video display or a sheet of printer paper), a point is signified by two numbers, generally the number pair (x, y), where x is the horizontal coordinate and y is the vertical coordinate. In Chapter 2, I discussed how the coordinates system in Windows Forms is defined, but the `Point` data type doesn't necessarily imply any particular coordinate system. You can use `Point` in any two-dimensional coordinate system.

The `Point` structure has two read-write properties, named `X` and `Y`, which are defined as 32-bit integers. `X` and `Y` can be negative. Even though `Point` is a `struct` rather than a `class`, you can't just define a variable of type `Point` and then assign values to the two properties:

```csharp
Point pt;
pt.X = 23;          // Compiler error here!
pt.Y = 47;
```

You'll get a message from the compiler complaining about the "use of unassigned local variable." You still need to use `new` to create an instance of a structure, just as with a class. The declaration

```csharp
Point pt = new Point();
```

results in both the `X` and `Y` properties being initialized to 0. Then you can set the `X` and `Y` properties to explicit values:

```csharp
pt.X = 23;
pt.Y = 47;
```

Or you can use this declaration to initialize the values:

```csharp
Point pt = new Point(34, 55);
```

Or, in a rare instance of bit packing in the .NET Framework, you can specify the two coordinates as 16-bit values stuck together in a 32-bit integer, as here:

```csharp
Point pt = new Point(0x01000010);
```
This declaration results in the X property being set to 16 (0x0010) and the Y property to 256 (0x0100). I don't suggest you begin treating points as single 32-bit integers; this declaration is mostly for the benefit of people who must continue to use Win32 API functions, which sometimes involve packed coordinates.

The only time you can get away without using new is when you use a method, property, or field that returns a Point. Actually, the Point structure has one such member itself. It's a static field named Empty:

Point pt = Point.Empty;

Notice the use of the capitalized Point on the right to indicate the Point structure itself rather than an instance of the Point structure. You need to reference the Point class because the Empty field is static. This statement results in the X and Y properties being initialized to 0. Point also has a read-only property named IsEmpty that returns true if both X and Y equal 0.

Here's a complete list of the Point properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>X</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Y</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>IsEmpty</td>
<td>get</td>
</tr>
</tbody>
</table>

Point inherits the GetType method from Object, overrides the GetHashCode, ToString, and Equals methods from Object by way of ValueType, and implements a method named Offset on its own. Here's a complete list of the public instance (that is, nonstatic) methods of Point:

Point Instance Methods

Type GetType()
int GetHashCode()
string ToString()
bool Equals(Point point)
void Offset(int dx, int dy)

There are three static methods of Point that I'll discuss shortly.

The ToString method converts a Point object to a readable character string. For example, after the statements

Point pt = new Point(5, 201);
string str = pt.ToString();

the str variable is set to the text string \{X=5,Y=201\}. The ToString method is called by Console.Write, Console.WriteLine, and String.Format to convert objects to strings.

The Equals method tests whether one point is equal to another, as in the statement

if (pt1.Equals(pt2))

Equality is defined to mean that the X and Y properties of the two Point objects are both equal to each other. More conveniently, in C#, you can also use the equals operator:

if (pt1 == pt2)

The Equals method is provided for languages that don't support an equals operator. You can also use the inequality operator with Point structures:

if (pt1 != pt2)
The *Offset* method

```
pt.Offset(21, -12);
```

is basically the same as adding the two offsets to the properties:

```
pt.X += 21;
pt.Y += -12;
```

### Arrays of Points

Arrays of *Point* structures are common in programming for graphical environments. For example, an array of *Point* structures could represent a complex curve or the locations of buttons on a calculator. To create an array of, say, 23 *Point* structures, you can use the following statement:

```
Point[] apt = new Point[23];
```

C# uses zero-based indexing for arrays, so the valid array elements are `apt[0]` through `apt[22]`. When you allocate an array of structures, each of the elements is initialized to the point (0, 0).

It's possible to initialize the array elements when you create the array, but it requires a bit more typing than when initializing an array of structures in C:

```
Point[] apt = new Point[3] { new Point(25, 50),
                           new Point(43, 32),
                           new Point(27, 8) };
```

You must have exactly as many initializers as the dimension of the array. Indeed, you don't need to include the dimension:

```
Point[] apt = new Point[] { new Point(25, 50),
                           new Point(43, 32),
                           new Point(27, 8) };
```

And you can even leave out the first `new` expression:

```
Point[] apt = { new Point(25, 50),
               new Point(43, 32),
               new Point(27, 8) };
```

### The Size Structure

The *Size* structure is very much like *Point*, but instead of the *X* and *Y* properties, it has *Width* and *Height* properties:

<table>
<thead>
<tr>
<th><strong>Size Properties</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><code>int</code></td>
</tr>
<tr>
<td><code>int</code></td>
</tr>
<tr>
<td><code>bool</code></td>
</tr>
</tbody>
</table>

You can create a new *Size* structure the same way you create a *Point* structure:

```
Size size = new Size(15, 20);
```

The *Width* and *Height* properties of the *Size* structure can be negative. (You'll start to see the reasoning behind this when we delve into rectangles later in this chapter.)

The *Point* and *Size* structures are so similar that they can be constructed from each other. Here's a complete list of the *Point* constructors:

### Point Constructors

```
Point()
```
Point(int xyPacked)
Point(int x, int y)
Point(Size size)

And here's a complete list of the Size constructors:

**Size Constructors**

- Size()
- Size(int width, int height)
- Size(Point point)

You can also cast a Point object to a Size object and vice versa:

```csharp
pt = (Point) size;
size = (Size) pt;
```

The Size structure overloads the addition and subtraction operators so you can add two sizes, as in

```csharp
size3 = size1 + size2;
```

or subtract two sizes, as here:

```csharp
size2 = size3 - size1;
```

What's going on here is what you'd expect: the Width and Height properties are being added or subtracted individually. You can also use the compound assignment operators:

```csharp
size2 += size1;
size3 -= size2;
```

The Point structure also overloads addition and subtraction, but I didn't mention these operations earlier because they also involve a Size structure. Under the assumption that it makes no sense to add two points together, the only objects you can add to or subtract from Point objects are Size objects.

```csharp
pt2 = pt1 + size;
pt3 = pt2 - size;
pt += size;
pt -= size;
```

**The Float Versions**

C# supports two floating-point data types, float and double. The double data type is rarely used in Windows Forms or GDI+, but float shows up a lot. You may wonder why you need floating-point coordinates when drawing in units of pixels, but you'll discover in Chapter 7 that you can use coordinate systems based on units other than pixels.

The PointF structure is very much like the Point structure except that the X and Y properties are float values rather than int. Similarly, the SizeF structure is very much like Size except that Width and Height are float values. The PointF and SizeF structures support the addition, subtraction, equality, and inequality operators just as Point and Size do.

Here's a mistake I make about once a week:

```csharp
PointF ptf = new PointF();
ptf.X = 2.5;                // Error!
ptf.Y = 3E-2;               // Error!
```
The compiler assumes the values are *double* rather than *float*, and as the compiler will remind you, "Literal of type double cannot be implicitly converted to type 'float'." One solution is to cast the values to *float* like so:

```csharp
ptf.X = (float)2.5;
ptf.Y = (float)3E-2;
```

But an easier method is to follow the literal with an *f* (for *float*):

```csharp
ptf.X = 2.5f;
ptf.Y = 3E-2f;
```

Like *Point*, the *PointF* structure implements a constructor that lets you initialize it during creation:

```csharp
PointF ptf = new PointF(2.5f, 3E-2f);
```

The *PointF* structure does not include an *Offset* method.

Integer values *can* be implicitly converted to *float*, so you can assign integers to the floating-point fields:

```csharp
PointF ptf = new PointF(127, 42);
```

You can cast a *Point* to a *PointF*:

```csharp
ptf = (PointF)pt;
```

However, you can't cast a *PointF* to a *Point*:

```csharp
pt = (Point)ptf;     // Error!
```

Instead, to convert a *PointF* to a *Point*, you must use one of the static methods provided for that purpose by the *Point* structure:

**Point Static Methods**

- `Point.Round(PointF ptf)`
- `Point.Truncate(PointF ptf)`
- `Point.Ceiling(PointF ptf)`

For example,

```csharp
pt = Point.Round(ptf);
```

*Round* must be preceded by the structure name because it's a static method.

The *Round* method rounds the X and Y properties to the nearest integer, and to the nearest positive integer for fractional parts of 0.5. The *Truncate* method essentially strips the fractional part and rounds toward 0. For example, coordinates of 0.9 and −0.9 both become 0. The *Ceiling* method rounds toward the next highest integer, that is, 0.9 becomes 1 and −0.9 becomes 0.

Similarly, you can cast a *Size* to a *SizeF*, but you should use the following methods to convert a *SizeF* to a *Size*:

**Size Static Methods**

- `Size.Round(SizeF sizef)`
- `Size.Truncate(SizeF sizef)`
- `Size.Ceiling(SizeF sizef)`
The `SizeF` structure also includes the following two instance methods, the only instance methods that `SizeF` doesn't inherit or override:

**SizeF Instance Methods**

- `PointF ToPointF()`
- `Size ToSize()`

The `ToSize` method is equivalent to the `Truncate` method.

Oddly enough, while you can cast between `Point` and `Size`, from `Point` to `PointF`, from `Size` to `SizeF`, and from `SizeF` to `PointF`, you can't cast from `PointF` to `SizeF`. However, `PointF` provides a constructor that takes a `SizeF` argument. Here, for comparison purposes, is a complete list of the constructors for the four structures:

**Constructor Comparisons**

<table>
<thead>
<tr>
<th></th>
<th><code>Point</code></th>
<th><code>PointF</code></th>
<th><code>Size</code></th>
<th><code>SizeF</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>(void)</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>(x, y)</td>
<td>(x, y)</td>
<td>(cx, cy)</td>
<td>(cx, cy)</td>
<td></td>
</tr>
<tr>
<td>(sz)</td>
<td>(point)</td>
<td>(pointf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(xyPacked)</td>
<td></td>
<td>(sizef)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A Rectangle Is a Point and a Size**

The `Rectangle` structure defines a rectangle as a combination of a `Point` and a `Size`. The idea here is that the `Point` refers to the location of the upper left corner of the rectangle and the `Size` is the width and height of this rectangle—which is not to say that the width and height must be non-negative. The `Rectangle` structure itself imposes no such restriction. However, as we'll explore in Chapters 4 and 5, the `Rectangle` structure is used in some drawing functions, and those functions require non-negative widths and heights. The `Rectangle` structure has two constructors:

**Rectangle Constructors**

- `Rectangle(Point pt, Size size)`
- `Rectangle(int x, int y, int width, int height)`

Veteran Windows programmers: Watch out for that last constructor, and in other places where you specify a rectangle using four numbers: the second two numbers are not the coordinates of the bottom right corner of the rectangle!

There's also a `RectangleF` structure with the following constructors:

**RectangleF Constructors**

- `RectangleF(PointF ptf, SizeF sizef)`
- `RectangleF(float x, float y, float width, float height)`

As you can see, the constructors for `Rectangle` and `RectangleF` are identical except for the data types.

In fact, the entire implementations of the `Rectangle` and `RectangleF` structures are so similar that you'll swear a template was somehow involved. Aside from the data types, the only difference is that
the `RectangleF` structure defines a cast from a `Rectangle` to a `RectangleF`, while the `Rectangle` structure defines three static (and by now familiar) methods that let you convert from a `RectangleF` to a `Rectangle`:

**Rectangle Static Methods (selection)**

- `Rectangle Round(RectangleF rectf)`
- `Rectangle Truncate(RectangleF rectf)`
- `Rectangle Ceiling(RectangleF rectf)`

So from here on, I'll refer to the `Rectangle` structure only, but everything I say applies to `RectangleF` as well. The data types associated with `RectangleF` are, of course, `float`, `PointF`, and `SizeF` rather than `int`, `Point`, and `Size`.

### Rectangle Properties and Methods

The `Rectangle` structure defines a host of properties that give you information in whatever way you want:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Point</code></td>
<td>Location</td>
<td>get/set</td>
</tr>
<tr>
<td><code>Size</code></td>
<td>Size</td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td>X</td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Y</td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Width</td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Height</td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Left</td>
<td>get</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Top</td>
<td>get</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Right</td>
<td>get</td>
</tr>
<tr>
<td><code>int</code></td>
<td>Bottom</td>
<td>get</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>IsEmpty</td>
<td>get</td>
</tr>
</tbody>
</table>

The `Left` property returns the same value as `X`; the `Top` property returns the same value as `Y`. The `Right` property returns the sum of `X` and `Width`; and the `Bottom` property returns the sum of `Y` and `Height`, even if `Width` and `Height` are negative. In other words, `Left` can be greater than `Right`, and `Bottom` can be greater than `Top`.

The only operators overloaded by `Rectangle` are equality (==) and inequality (!=). `Rectangle` also implements an `Equals` method. Although addition and subtraction are not allowed on `Rectangle` structures, several methods do allow you to manipulate `Rectangle` structures in various ways or to create new `Rectangle` structures from existing ones.

For Windows programmers accustomed to thinking in terms of upper left and lower right, there's a static method that creates a `Rectangle` from those two coordinates:

**Rectangle Methods (selection)**

- `static Rectangle FromLTRB(int xLeft, int yTop, int xRight, int yBottom)`
Because this is a \textit{static} method, the method name must be prefaced with the structure name. The method returns a newly created \textit{Rectangle} object. The call
\begin{verbatim}
rect = Rectangle.FromLTRB(x1, y1, x2, y2);
\end{verbatim}
is equivalent to
\begin{verbatim}
rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);
\end{verbatim}
The \textit{Offset} and \textit{Inflate} methods manipulate a \textit{Rectangle} structure and compensate for the lack of addition and subtraction operators:

\begin{verbatim}
Rectangle Methods (selection)
\end{verbatim}

\begin{verbatim}
void Offset(int x, int y)
void Offset(Point)
void Inflate(int x, int y)
void Inflate(Size size)
static Rectangle Inflate(Rectangle rect, int x, int y)
\end{verbatim}

The \textit{Offset} method shifts a rectangle to another location. The method call
\begin{verbatim}
rect.Offset(23, -46);
\end{verbatim}
is equivalent to
\begin{verbatim}
rect.X += 23;
rect.Y += -46;
\end{verbatim}
or
\begin{verbatim}
rect.Location += new Size(23, -46);
\end{verbatim}

An overloaded version of \textit{Offset} takes a \textit{Point} argument (which should probably be a \textit{Size}):
\begin{verbatim}
rect.Offset(pt);
\end{verbatim}
That call is equivalent to
\begin{verbatim}
rect.X += pt.X;
rect.Y += pt.Y;
\end{verbatim}
or
\begin{verbatim}
rect.Location += (Size)pt;
\end{verbatim}
The \textit{Inflate} method affects both the location and size of the \textit{Rectangle}:
\begin{verbatim}
rect.Inflate(x, y);
\end{verbatim}
is equivalent to
\begin{verbatim}
rect.X -= x;
rect.Y -= y;
rect.Width  += 2 * x;
rect.Height += 2 * y;
\end{verbatim}
Thus, the rectangle gets larger (or smaller, if the arguments are negative) in all directions. The center of the rectangle remains the same. An overload to \textit{Inflate} uses a \textit{Size} object to provide the two values. A \textit{static} version creates a new inflated \textit{Rectangle} object from an existing \textit{Rectangle} object.

The following methods perform unions and intersections of pairs of \textit{Rectangle} objects:

\begin{verbatim}
Rectangle Methods (selection)
\end{verbatim}
The `Intersect` method has an overload that isn't `static`. You use it like this:
```csharp
rect1.Intersect(rect2);
```
This expression is equivalent to
```csharp
rect1 = Rectangle.Intersect(rect1, rect2);
```
The remaining methods unique to `Rectangle` return `bool` values:

**Rectangle Methods (selection)**

- `bool Contains(Point)`
- `bool Contains(int x, int y)`
- `bool Contains(Rectangle rect)`
- `bool IntersectsWith(Rectangle rect)`

Finally, both `Rectangle` and `RectangleF` override `ToString` in a useful manner, returning a string that looks something like this:
```
{x=12,Y=5,Width=30,Height=10}
```

**A Nice-Sized Form**

How large is your form? This is not a personal question! When a program creates a form, the form has a specific size and occupies a specific location on the screen. The size and location are not fixed, however: If the form has a sizing border, the user can drag that border to make the form a different size. If the form has a caption bar, the user can move the form to another location on the screen. It might be helpful for a program to know how large its form is. Knowing exactly where the form is located on the screen is less useful but not totally irrelevant.

The `Form` class has no fewer than 13 properties—most of them inherited from the `Control` class—that reveal this information. With just a couple exceptions, these properties are also writable and allow a program to change the size and location of its form. We saw an example of such changes in the FormProperties program in Chapter 2.

Here are nine properties (all of which `Form` inherits from `Control`) that indicate the size of the form and its location on the screen:

**Control Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>Location</td>
<td>get/set</td>
<td>Relative to screen</td>
</tr>
<tr>
<td>Size</td>
<td>Size</td>
<td>get/set</td>
<td>Size of full form</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Bounds</td>
<td>get/set</td>
<td>Equals <code>Rectangle(Location, Size)</code></td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get/set</td>
<td>Equals <code>Size.Width</code></td>
</tr>
<tr>
<td>int</td>
<td>Height</td>
<td>get/set</td>
<td>Equals <code>Size.Height</code></td>
</tr>
<tr>
<td>int</td>
<td>Left</td>
<td>get/set</td>
<td>Equals <code>Location.X</code></td>
</tr>
<tr>
<td>int</td>
<td>Top</td>
<td>get/set</td>
<td>Equals <code>Location.Y</code></td>
</tr>
</tbody>
</table>
Basically, what we're dealing with here can be reduced to four numbers: the x and y coordinates of the upper left corner of the form relative to the upper left corner of the video display, and the height and width of the form. I suspect that the only reason `Right` and `Bottom` are read-only is because it's not clear what should happen when you set them. Do you want to move the form or make it a different size?

Although you can set `Width` and `Height` to any values you want, the system imposes some limits. The lower limits are values sufficient to display enough of the caption bar to read some of the text. The upper limits prevent the form from being larger than if it were maximized to the size of the screen.

Don't do this, however:

```csharp
Size.Width *= 2;
```

That's setting a property of a property. For reasons beyond the comprehension of people who don't write compilers, it's not allowed.

Two additional size-related and location-related properties are defined in the `Form` class:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Point</code></td>
<td><code>DesktopLocation</code></td>
<td>get/set</td>
<td></td>
</tr>
<tr>
<td><code>Rectangle</code></td>
<td><code>DesktopBounds</code></td>
<td>get/set</td>
<td></td>
</tr>
</tbody>
</table>

These are similar to the `Location` and `Bounds` properties but take the Windows taskbar into account. The desktop is that part of the screen not occupied by the taskbar. If the taskbar is positioned at the left, `DesktopLocation.X` will be less than `Location.X`; if the taskbar is at the top, `DesktopLocation.Y` will be less than `Location.Y`. `DesktopBounds` is based on `DesktopLocation` and the normal `Size` property, which isn't affected by the position of the taskbar.

**The Form and the Client**

The dimensions I've been presenting refer to the entire form, including the border and the caption bar. In most cases, an application is primarily interested in the size of the form's client area. The **client area** is the internal area of the form upon which the application is free to draw during the form's `Paint` event or otherwise decorate with controls. The client area excludes the form's caption bar and any border the form may have. If the form includes a menu bar beneath its caption bar (I'll discuss how to add menu bars in Chapter 14), the client area also excludes the area occupied by that menu bar. If the form displays scroll bars at the right and bottom of the window (I'll show you how to do this in Chapter 4), the client area also excludes these scroll bars.

The `Form` class has just two properties (also first implemented in `Control`) that pertain to the size of the client area:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Size</code></td>
<td><code>ClientSize</code></td>
<td>get/set</td>
<td></td>
</tr>
<tr>
<td><code>Rectangle</code></td>
<td><code>ClientRect</code></td>
<td>get</td>
<td></td>
</tr>
</tbody>
</table>

The `ClientSize` property indicates the pixel width and height of the client area. The `ClientRect` property supplies no additional information because the X and Y properties of `ClientRect` are always 0! `ClientRect` is sometimes useful in methods that require `Rectangle` arguments. The last two programs in this chapter illustrate how `Rectangle` can be used for this purpose.
Again, avoid setting a property of a property. This assignment won't work:

```csharp
ClientSize.Width += 100; // Won't work!
```

Do this instead:

```csharp
ClientSize = new Size(ClientSize.Width + 100, ClientSize.Height);
```

Or do this:

```csharp
ClientSize += new Size(100, 0);
```

The following program displays all thirteen of the size and location properties in its client area.

**FormSize.cs**

```csharp
// FormSize.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class FormSize: Form
{
    public static void Main()
    {
        Application.Run(new FormSize());
    }
    public FormSize()
    {
        Text = "Form Size";
        BackColor = Color.White;
    }
    protected override void OnMove(EventArgs ea)
    {
        Invalidate();
    }
    protected override void OnResize(EventArgs ea)
    {
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        string str = "Location: " + Location + "\n" + "Size: " + Size + "\n" + "Bounds: " + Bounds + "\n" + "Width: " + Width + "\n" + "Height: " + Height + "\n" + "Left: " + Left + "\n" + "Top: " + Top + "\n";
    }
}```
This innocently intended program introduces a couple things we haven't seen before. First, notice that I'm overriding the `OnMove` and `OnResize` methods. As the `On` prefixes indicate, these methods are associated with events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>OnMove</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Resize</td>
<td>OnResize</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

These methods are called when the form is moved or resized. In real-life programming, the `OnMove` method is almost never overridden. Programs usually don't care where they are located on the screen. Overriding the `OnResize` method is quite common, however. We'll see many examples of `OnResize` used in the chapters ahead.

In response to both these events, I'm calling the simplest of the six overloads of the `Invalidate` method:

### Control Invalidate Methods

```csharp
void Invalidate();
void Invalidate(Rectangle rectInvalid);
void Invalidate(bool bIncludeChildren);
void Invalidate(Rectangle rectInvalid, bool bIncludeChildren);
void Invalidate(Region rgnInvalid);
void Invalidate(Region rgnInvalid, bool bIncludeChildren);
```

What this method does is invalidate the entire client area, or a rectangular or nonrectangular subset of it, with or without invalidating any child controls that may be located on it. Invalidating the client area informs Windows that what's on there is no longer valid. Eventually, your form will receive a call to `OnPaint` so that you have an opportunity to repaint the client area. Calling `Invalidate` is the standard way in which a program forces a repaint.

The `OnPaint` call doesn't occur right away. Whatever event the form is currently processing (in this case, the `Resize` or `Move` event) must be completed first, and even then, if other events are pending (such as keyboard or mouse input), the `OnPaint` call must wait. If you want to update your client area immediately, follow the `Invalidate` call with a call to the `Control` object's `Update` method:

### Control Methods (selection)

```csharp
void Update();
```
This will cause an immediate call to `OnPaint`; after `OnPaint` returns, `Update` will return.

Something else the FormSize program demonstrates is string concatenation with `Point`, `Size`, and `Rectangle` objects. When you put a property or another object in a string as we did in this program, the object's `ToString` method is invoked.

I've also taken advantage of the fact that the `DrawString` method recognizes ASCII line feed characters and correctly spaces successive lines of text. As it is, the output is a bit sloppy:

![Form Size window with properties](image)

We'll learn how to put text into nice neat columns in the next chapter.

There is a 14th property of the `Form` object inherited from `Control` that is related to the size of the client area:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>DisplayRectangle</td>
<td>get</td>
</tr>
</tbody>
</table>

By default, this property is the same as `ClientRect` and doesn't change unless you start putting controls on the client area.

### Point Conversions

As you saw in a couple programs in Chapter 2, when you draw graphics on your form, you're using a coordinate system that is relative to the upper left corner of the client area. These coordinates are referred to as client area coordinates. Earlier, when I discussed the location of the form relative to the screen, I implicitly introduced another coordinate system. This coordinate system is relative to the upper left corner of the screen, and such coordinates are called screen coordinates. Often desktop coordinates are the same as screen coordinates, but not if the Windows taskbar is on the top or left edge of the screen. Finally, form coordinates are relative to the upper left corner of the form, which is usually the corner of the form's sizing border.

The `Location` property refers to a point in screen coordinates that is equivalent to the point (0, 0) in form coordinates. Thus, this property allows an application to convert between points in these two coordinate systems. Symbolically,

\[
x_{\text{screen}} = x_{\text{form}} + Location.X \\
y_{\text{screen}} = y_{\text{form}} + Location.Y
\]

Similarly, the form's `DesktopLocation` property allows a program to convert between desktop coordinates and form coordinates:

\[
x_{\text{desktop}} = x_{\text{form}} + DesktopLocation.X \\
y_{\text{desktop}} = y_{\text{form}} + DesktopLocation.Y
\]

With some simple algebraic manipulation, you can also convert between desktop coordinates and screen coordinates:

\[
x_{\text{desktop}} = x_{\text{screen}} + DesktopLocation.X - Location.X \\
y_{\text{desktop}} = y_{\text{screen}} + DesktopLocation.Y - Location.Y
\]
There aren't any similar properties of Form that allow an application to convert between client area coordinates and any of the other three coordinate systems. It's possible, by using the CaptionHeight property of the SystemInformation class, to obtain the height of the standard caption bar and then to obtain the width of the sizing border by comparing ClientSize with the form's total Size (less the caption bar height), but that's more work than you should have to do.

Fortunately, the Form class contains two methods that convert points directly between screen coordinates and client area coordinates:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point PointToClient(Point ptScreen)</td>
<td>Converts screen coordinates to client</td>
</tr>
<tr>
<td>Point PointToScreen(Point ptClient)</td>
<td>Converts client coordinates to screen</td>
</tr>
</tbody>
</table>

The Point passed as an argument to these methods remains unchanged; the methods return a Point containing the converted points. For example, the call

```
Point pt = PointToScreen(Point.Empty);
```

returns the location of the upper left corner of the client area in screen coordinates.

The Form class also supports two additional conversion methods that work with Rectangle objects rather than Point objects:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle RectangleToClient(Rectangle rectScreen)</td>
<td>Converts screen coordinates to client</td>
</tr>
<tr>
<td>Rectangle RectangleToScreen(Rectangle rectClient)</td>
<td>Converts client coordinates to screen</td>
</tr>
</tbody>
</table>

These methods don't provide any additional information than PointToClient and PointToScreen because the Size property of the Rectangle object remains unaffected by the conversion.

The Color Structure

The human eye perceives electromagnetic radiation in the range of about 430 to 750 terahertz, corresponding to wavelengths between 700 and 400 nanometers. Electromagnetic radiation in this range is known as visible light. If the light isn't very strong, the 120 million rods in the retina of the human eye respond to the light's intensity. Stronger light affects the 7 million cones, which come in three different types, each of which responds to a different range of wavelengths. The varying degrees of excitation of these cones is the phenomena we call color, and the three ranges of wavelengths correspond to our concepts of red, green, and blue.

Because very little data is required to specify a color, color is a good candidate for a structure rather than a class and, indeed, Color is another important structure in the System.Drawing namespace.

Color in Windows Forms is based on an ARGB (alpha-red-green-blue) model. Colors themselves are generally defined by single-byte values of red, green, and blue. The alpha channel determines the transparency of the color. Alpha values range from 0 for transparent to 0xFF for opaque.

The Color structure has only a default constructor, which you can use like so:

```
Color color = new Color();
```

You almost surely won't be using code like this, however, because it would create an empty color (transparent black) and there's no way to change the properties of that color. Instead, you'll be obtaining color objects by using one of the static methods or properties defined in Color for that purpose.
The static properties in Color are quite valuable, for there are no fewer than 141 of them.

The 141 Color Names

The Color structure has 140 static read-only properties that are actual names of colors ranging (in alphabetical order) from AliceBlue[1] to YellowGreen. Only a couple of the names (Magenta and Fuchsia, for example) represent identical colors; most of them are unique colors. The Color class also has a 141st property, named Transparent, that represents a transparent color. The following table shows some of the 141 properties in the Color class. I haven't included all the properties because such a listing would have run to four pages.

<table>
<thead>
<tr>
<th>Color Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>Color</td>
</tr>
</tbody>
</table>

You can find a complete list of the 140 standard (and sometimes whimsically named) colors on the inside back cover of this book.

Where did these colors come from? They originated in the X Window System, X11R3 (version 11, revision 3), which is a graphical user interface developed at MIT for UNIX. More recently, these colors were considered for inclusion in the Cascading Style Sheets (CSS) standard from the World Wide Web Consortium (W3C), but they were removed before the specification was finalized. Nevertheless, these 140 colors have become a de facto standard in HTML, being supported by recent versions of both Microsoft Internet Explorer and Netscape Navigator.

Whenever you need a Color object, you can just use Color.Red (or whatever color you want) and it'll work. I've already done this in some of the programs, when setting theBackColor property of a form and as an argument to the Clear method of the Graphics class.

To create a color based on the red, green, blue, and alpha components, you can use the following Color.FromArgb static methods, each of which returns a Color object:

**Color_FromArgb Static Methods**

- Color Color.FromArgb(int r, int g, int b)
- Color Color.FromArgb(int a, int r, int g, int b)
- Color Color.FromArgb(int a, Color color)
- Color Color.FromArgb(int argbPacked)

I'll use the first of these methods in the RandomClear program later in this chapter.

[1] AliceBlue gets its name from Alice Roosevelt (1884–1980), who was a spirited teenager when her father became president in 1901 and whose favorite color was immortalized in fashion and song. See [www.theodoreroosevelt.org/life/familytree/AliceLongworth.htm](http://www.theodoreroosevelt.org/life/familytree/AliceLongworth.htm) and [www.theodoreroosevelt.org/life/alicelblue.htm](http://www.theodoreroosevelt.org/life/alicelblue.htm) for more details.

**Pens and Brushes**

Color objects by themselves aren't used much in Windows Forms. You've seen how you can set the BackColor property to a Color object. There's also a ForeColor property you can set likewise. The Clear method in the Graphics class also takes a Color argument, but that's an exception.
Most of the other Graphics drawing methods don't involve Color arguments. When you draw lines or curves (which you'll start doing in Chapter 5), you use an object of type Pen, and when you draw filled areas and text, you specify an object of type Brush. Of course, pens and brushes themselves are specified using color, but other characteristics are often involved as well.

You create a pen using one of the four constructors of the Pen class. The simplest of these constructors creates a Pen object based on a Color object:

```java
Pen pen = new Pen(clr);
```

If you want to create a pen based on one of the predefined colors, you don't need to do this:

```java
Pen pen = new Pen(Color.RosyBrown);
```

It's better to use the Pens class instead. (Notice the plural on the class name.) Pens consists solely of 141 static read-only properties, each of which returns an object of type Pen. Aside from the return type, these properties are identical to the 141 Color properties.

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pen</td>
<td>AliceBlue</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>AntiqueWhite</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>Yellow</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>YellowGreen</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>Transparent</td>
<td>get</td>
</tr>
</tbody>
</table>

You'll learn more about the Pen class in Chapter 5, and we'll really dig into the details of it in Chapter 17.

When you draw text or filled areas, you specify a Brush object. The Brush class itself is abstract, which means you can't create an instance of it. Brush is instead the parent class for five other classes: SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush, and PathGradientBrush. We'll go over brushes in depth in Chapter 17. For now, be aware that you can create a brush of a solid color like so:

```java
Brush brush = new SolidBrush(clr);
```

You can assign the result to an object of type Brush because SolidBrush is inherited from Brush.

As with the Pen class, using one of the static Color properties in SolidBrush is unnecessary because the Brushes class (notice the plural again) consists solely of—that's right!—141 static read-only properties that return objects of type Brush.

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brush</td>
<td>AliceBlue</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>AntiqueWhite</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Yellow</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>YellowGreen</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Transparent</td>
<td>get</td>
</tr>
</tbody>
</table>

This is the class we used in Chapter 2 to provide a black brush (Brushes.Black) for the DrawString method. I mentioned at the time that you should use Brushes.Black for text only when you're assured...
that the background of your form isn't also black. One way to do this is to set the form's background color explicitly:

```csharp
BackColor = Color.White;
```

## System Colors

The reason `Brushes.Black` isn't a good idea for text is that it's possible for a Windows user to invoke the Display Properties dialog box (either from Control Panel or by right-clicking the desktop), select the Appearance tab, and choose a color scheme, such as High Contrast Black, in which the background color of windows and controls is black. People with poor eyesight or color blindness often use such high-contrast color schemes, and you're definitely not helping them if you display your text in black as well!

Welcome to the world of system colors, which are probably more correctly called user-preference colors. Using the Display Properties dialog box, users can select their own color schemes. Windows itself maintains 29 user-settable colors that it employs to color different components of the user interface. Twenty-six of these colors are exposed in the Windows Forms framework.

You can obtain these color values from the `SystemColors` class, which consists solely of 26 read-only properties, each of which returns a `Color` object:

### SystemColors Static Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>ActiveBorder</td>
<td>get</td>
<td>Border of active window</td>
</tr>
<tr>
<td>Color</td>
<td>ActiveCaption</td>
<td>get</td>
<td>Caption bar of active window</td>
</tr>
<tr>
<td>Color</td>
<td>ActiveCaptionText</td>
<td>get</td>
<td>Caption bar text of active window</td>
</tr>
<tr>
<td>Color</td>
<td>AppWorkspace</td>
<td>get</td>
<td>Workspace background in a multiple-document interface (MDI)</td>
</tr>
<tr>
<td>Color</td>
<td>Control</td>
<td>get</td>
<td>Background color of controls</td>
</tr>
<tr>
<td>Color</td>
<td>ControlDark</td>
<td>get</td>
<td>Shadows of 3D controls</td>
</tr>
<tr>
<td>Color</td>
<td>ControlDarkDark</td>
<td>get</td>
<td>Dark shadows of 3D controls</td>
</tr>
<tr>
<td>Color</td>
<td>ControlLight</td>
<td>get</td>
<td>Highlights of 3D controls</td>
</tr>
<tr>
<td>Color</td>
<td>ControlLightLight</td>
<td>get</td>
<td>Light highlights of 3D controls</td>
</tr>
<tr>
<td>Color</td>
<td>ControlText</td>
<td>get</td>
<td>Text color of controls</td>
</tr>
<tr>
<td>Color</td>
<td>Desktop</td>
<td>get</td>
<td>Windows desktop</td>
</tr>
<tr>
<td>Color</td>
<td>GrayText</td>
<td>get</td>
<td>Disabled text</td>
</tr>
<tr>
<td>Color</td>
<td>Highlight</td>
<td>get</td>
<td>Highlighted text background</td>
</tr>
<tr>
<td>Color</td>
<td>HighlightText</td>
<td>get</td>
<td>Highlighted text</td>
</tr>
<tr>
<td>Color</td>
<td>HotTrack</td>
<td>get</td>
<td>Hot track</td>
</tr>
<tr>
<td>Color</td>
<td>InactiveBorder</td>
<td>get</td>
<td>Border of inactive windows</td>
</tr>
<tr>
<td>Color</td>
<td>InactiveCaption</td>
<td>get</td>
<td>Caption bar of inactive windows</td>
</tr>
<tr>
<td>Color</td>
<td>InactiveCaptionText</td>
<td>get</td>
<td>Caption bar text of inactive windows</td>
</tr>
<tr>
<td>Color</td>
<td>Info</td>
<td>get</td>
<td>ToolTip background</td>
</tr>
<tr>
<td>Color</td>
<td>InfoText</td>
<td>get</td>
<td>ToolTip text</td>
</tr>
<tr>
<td>Color</td>
<td>Menu</td>
<td>get</td>
<td>Menu background</td>
</tr>
<tr>
<td>Color</td>
<td>MenuText</td>
<td>get</td>
<td>Menu text</td>
</tr>
<tr>
<td>Color</td>
<td>ScrollBar</td>
<td>get</td>
<td>Scroll bar background</td>
</tr>
</tbody>
</table>
### SystemColors Static Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Window</td>
<td>get</td>
<td>Window background</td>
</tr>
<tr>
<td>Color</td>
<td>WindowFrame</td>
<td>get</td>
<td>Thin window frame</td>
</tr>
<tr>
<td>Color</td>
<td>WindowText</td>
<td>get</td>
<td>Window text</td>
</tr>
</tbody>
</table>

You could create a Pen or a Brush from one of these colors like so:

```csharp
Pen pen = new Pen(SystemColor.ControlText);
Brush brush = new SolidBrush(SystemColor.ControlText);
```

It's usually not necessary to do this, however, because the `System.Drawing` namespace also includes a `SystemPens` class and a `SystemBrushes` class. `SystemPens` has 15 static read-only properties that return objects of type `Pen`:

### SystemPens Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pen</td>
<td>ActiveCaptionText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>Control</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>ControlDark</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>ControlDarkDark</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>ControlLight</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>ControlLightLight</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>ControlText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>GrayText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>Highlight</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>HighlightText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>InactiveCaptionText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>InfoText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>MenuText</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>WindowFrame</td>
<td>get</td>
</tr>
<tr>
<td>Pen</td>
<td>WindowText</td>
<td>get</td>
</tr>
</tbody>
</table>

The `SystemBrushes` class contains 21 static read-only properties that return objects of type `Brush`:

### SystemBrushes Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brush</td>
<td>ActiveBorder</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ActiveCaption</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ActiveCaptionText</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>AppWorkspace</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Control</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ControlDark</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ControlDarkDark</td>
<td>get</td>
</tr>
</tbody>
</table>
**SystemBrushes Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brush</td>
<td>ControlLight</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ControlLightLight</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ControlText</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Desktop</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Highlight</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>HighlightText</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>HotTrack</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>InactiveBorder</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>InactiveCaption</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Info</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Menu</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>ScrollBar</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>Window</td>
<td>get</td>
</tr>
<tr>
<td>Brush</td>
<td>WindowText</td>
<td>get</td>
</tr>
</tbody>
</table>

Strangely enough, not all the system colors that pertain to text are included in the `SystemBrushes` class. However, if you need a `Pen` or `Brush` that's not included in the `SystemPens` or `SystemBrushes` class, you can always create it using one of the `SystemColors` properties as an argument to one of the following static methods:

```csharp
Pen pen = SystemPens.FromSystemColor(SystemColor.ActiveBorder);
Brush brush = SystemBrushes.FromSystemColor(SystemColor.MenuText);
```

**The Known Colors**

The final big color list is the `KnownColor` enumeration that encompasses all the color names and all the system colors:

**KnownColor Enumeration**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveBorder</td>
<td>1</td>
<td>InactiveCaption</td>
<td>17</td>
</tr>
<tr>
<td>ActiveCaption</td>
<td>2</td>
<td>InactiveCaptionText</td>
<td>18</td>
</tr>
<tr>
<td>ActiveCaptionText</td>
<td>3</td>
<td>Info</td>
<td>19</td>
</tr>
<tr>
<td>AppWorkspace</td>
<td>4</td>
<td>InfoText</td>
<td>20</td>
</tr>
<tr>
<td>Control</td>
<td>5</td>
<td>Menu</td>
<td>21</td>
</tr>
<tr>
<td>ControlDark</td>
<td>6</td>
<td>MenuText</td>
<td>22</td>
</tr>
<tr>
<td>ControlDarkDark</td>
<td>7</td>
<td>ScrollBar</td>
<td>23</td>
</tr>
<tr>
<td>ControlLight</td>
<td>8</td>
<td>Window</td>
<td>24</td>
</tr>
<tr>
<td>ControlLightLight</td>
<td>9</td>
<td>WindowFrame</td>
<td>25</td>
</tr>
<tr>
<td>ControlText</td>
<td>10</td>
<td>WindowText</td>
<td>26</td>
</tr>
<tr>
<td>Desktop</td>
<td>11</td>
<td>Transparent</td>
<td>27</td>
</tr>
</tbody>
</table>
Although `KnownColor` is the third largest enumeration in the entire .NET Framework, it's not used for very much. The `Color` class has a static method that lets you create a color based on a `KnownColor` value, but if that's something you need, it probably makes more sense to use one of the static `Color` or `SystemColors` properties.

### What to Use?

Somewhere in the constructor for the `Control` class, the following code is probably executed:

```csharp
BackColor = SystemColors.Control;
ForeColor = SystemColors.ControlText;
```

What the `Control` and `ControlText` system colors are is entirely up to the user. Normally they're gray and black, respectively.

When a button control (for example) draws itself, it uses the `BackColor` property to color its background and the `ForeColor` property to display the button text. A `Form` object uses the `BackColor` property to erase the background of the client area but doesn't itself use the `ForeColor` property. That property is made available for applications inheriting or instantiating `Form`.

So, the question is, What brush should you be using to draw text? I think I've pretty well established that it's not `Brushes.Black`. A much better choice would be `SystemBrushes.ControlText`.

However, I'm not so sure that's optimum either. Consider this question: Why are the `BackColor` and `ForeColor` properties of `Form` set to the system colors used for controls? The answer is, because the Windows Forms developers have assumed that you'll be covering a form with controls or using a form for a dialog box.

If you're not putting controls on a form, though, and if you want your form to look like a regular old Windows program, you should be putting the following two statements in the constructor for your form:

```csharp
BackColor = SystemColors.Window;
ForeColor = SystemColors.WindowText;
```

In that case, the `DrawString` calls in your `OnPaint` code should use `SystemBrushes.WindowText` to be consistent.

But why write `OnPaint` code that's dependent on the way you set `BackColor` and `ForeColor` in the constructor? To write ideally generalized code, the brush you should use in your `DrawString` calls is new `SolidBrush(ForeColor)`

And that's the brush I'll be using for the remainder of this book whenever I want to display text in the user-preferred color.

Until I start creating controls on the surface of my forms, I'll also be setting the `BackColor` and `ForeColor` properties to `SystemColors.Window` and `SystemColors.WindowText` whenever I know I'll be drawing something that depends on those colors.

### Getting a Feel for Repaints
As you've seen, the background of a form is automatically colored by the property `BackColor`. You've also seen another way to recolor the background of a form: by using the `Clear` method of the `Graphics` class. `Clear` has one argument, which is a `Color` object:

```csharp
Graphics Methods (selection)

```void Clear(Color color)`

The RandomClear program randomly calculates a new color whenever its `OnPaint` method is called and uses the `Clear` method to display the new color.

`RandomClear.cs`

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class RandomClear: Form
{
    public static void Main()
    {
        Application.Run(new RandomClear());
    }
    public RandomClear()
    {
        Text = "Random Clear";
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Random rand = new Random();

        grfx.Clear(Color.FromArgb(rand.Next(256),
                                  rand.Next(256),
                                  rand.Next(256)));
    }
}
```

Run this program, and experiment with resizing the form. Think about what you're seeing: as you make the form larger, the newly uncovered strips on the right and bottom get a different color. Every new color you see represents a new call to `OnPaint`. Even though the `Clear` method is seemingly clearing the entire client area, however, it's actually restricted to the region that's newly invalid. (You'll notice that if you make the form smaller, the color doesn't change because there is no area of the client that hasn't remained valid.)

This behavior isn't always desirable. It could be that you're writing a program in which you want the entire client area to be invalidated whenever the size of the client area changes. I showed one way to do that in the FormSize program earlier in this chapter: override the `OnResize` method and put in an `Invalidate` call.
Another solution is to set the ResizeRedraw property to true in the form's constructor:

```csharp
ResizeRedraw = true;
```

The ResizeRedraw property causes the entire client area to be invalidated whenever its size changes. The following program demonstrates the difference.

**RandomClearResizeRedraw.cs**
```
//------------------------------------------------------
// RandomClearResizeRedraw.cs © 2001 by Charles Petzold
//------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RandomClearResizeRedraw: Form
{
    public static void Main()
    {
        Application.Run(new RandomClearResizeRedraw());
    }
    public RandomClearResizeRedraw()
    {
        Text = "Random Clear with Resize Redraw";
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Random rand = new Random();

        grfx.Clear(Color.FromArgb(rand.Next(256),
                                      rand.Next(256),
                                      rand.Next(256)));
    }
}
```

I'm a little hesitant about recommending that you put this ResizeRedraw assignment in every Windows Forms program you write, or even every sizable control. It probably shows up more in this book than in real life because I like to write programs that change their contents based on the size of the client area.

But keep this advice in mind: Whenever something on your form isn't being updated correctly, you should think about whether setting the ResizeRedraw property makes sense. And if you've already set ResizeRedraw, well, the problem is something else.

Before we leave this program, here's a little exercise. Put the following do-nothing OnResize override in RandomClearResizeRedraw:

```csharp
protected override void OnResize(EventArgs ea)
{
}
```
What you'll find is that the program now behaves exactly like RandomClear. Obviously, the `OnResize` method implemented in `Control` (which `Form` inherits) is responsible for invalidating the control depending on the style. `OnResize` probably contains some code that looks like this:

```csharp
if (ResizeRedraw)
    Invalidate();
```

For this reason and others, whenever you override the `OnResize` method, you should make a call to the `OnResize` method implemented in the base class:

```csharp
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    // Do what the program needs.
}
```

### Centering Hello World

Who was that kid in the back of the classroom who asked about centering text in a program's client area? We are now, at long last, ready to do it. Does such a program require setting the `ResizeRedraw` property? Yes, it certainly does, because what constitutes the center of the client area depends on the overall size of the client area.

One approach that might occur to you is to change the coordinate point in the `DrawString` function. Instead of using

```csharp
grfx.DrawString("Hello, world!", Font, brush, 0, 0);
```

you would use

```csharp
grfx.DrawString("Hello, world!", Font, brush,
               ClientSize.Width / 2, ClientSize.Height / 2);
```

That's a good start, but it's not quite right. Remember that the coordinate point passed to `DrawString` specifies the position of the upper left corner of the text string, so that's what will be positioned in the center of the client area. The text won't be in the center of the client area but will instead be situated in the upper left corner of the bottom right quadrant of the client area.

It's possible to alter this default behavior by using one of the overloaded versions of `DrawString`—a version that includes another argument along with the text string, font, brush, and starting position. The additional argument is an object of type `StringFormat`, the purpose of which is to specify in more detail how you want the text to be displayed.

An extensive discussion of `StringFormat` awaits us in Chapter 9. For now, let's just focus on the most-used facility of `StringFormat`, which is to change the default text alignment—the rule that says that the coordinate point passed to `DrawString` refers to the position where the upper left corner of the text is to be displayed.

To change the default text alignment, you must first create an object of type `StringFormat`:

```csharp
StringFormat strfmt = new StringFormat();
```

You can then set two properties of this object to specify the desired text alignment:

<table>
<thead>
<tr>
<th><code>StringFormat</code> Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td><code>StringAlignment</code></td>
</tr>
<tr>
<td><code>StringAlignment</code></td>
</tr>
</tbody>
</table>

Both these properties are of type `StringAlignment`, which is an enumeration consisting of three oddly named members:
### StringAlignment Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near</td>
<td>0</td>
<td>Usually left or top</td>
</tr>
<tr>
<td>Center</td>
<td>1</td>
<td>Always the center</td>
</tr>
<tr>
<td>Far</td>
<td>2</td>
<td>Usually right or bottom</td>
</tr>
</tbody>
</table>

Well, OK, so only two of the members are oddly named. Windows Forms has the ability to display text right-to-left, as is normal in some languages, or vertically, also normal in some languages. Near and Far are intended to be orientation-neutral, meaning "nearest to the beginning of the text" or "farthest from the beginning of the text."

If you know you'll be displaying left-to-right horizontally oriented text, you can think of `StringAlignment.Near` as meaning left and `StringAlignment.Far` as right when used with the `Alignment` property, and `StringAlignment.Near` as top and `StringAlignment.Far` as bottom when used with the `LineAlignment` property. Here's a program that uses all four combinations of these settings to display text strings in the four corners of the client area.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class FourCorners : Form
{
    public static void Main()
    {
        Application.Run(new FourCorners());
    }
    public FourCorners()
    {
        Text = "Four Corners Text Alignment";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Brush brush = new SolidBrush(ForeColor);
        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = StringAlignment.Near;
        strfmt.LineAlignment = StringAlignment.Near;
        grfx.DrawString("Upper left corner", Font, brush, 0, 0, strfmt);
    }
}
```

strfmt.Alignment     = StringAlignment.Far;
strfmt.LineAlignment = StringAlignment.Near;
grfx.DrawString("Upper right corner", Font, brush,
                ClientSize.Width, 0, strfmt);

strfmt.Alignment     = StringAlignment.Near;
strfmt.LineAlignment = StringAlignment.Far;
grfx.DrawString("Lower left corner", Font, brush,
                0, ClientSize.Height, strfmt);

strfmt.Alignment     = StringAlignment.Far;
strfmt.LineAlignment = StringAlignment.Far;
grfx.DrawString("Lower right corner", Font, brush,
                ClientSize.Width, ClientSize.Height, strfmt);
}
}

The coordinate points passed to the four **DrawString** calls refer to the four corners of the client area. Here's what it looks like:

![Four Corners Text Alignment](image)

If **StringFormat** were not used, however, only the first **DrawString** call would result in visible text. The text displayed by the other three calls would be positioned completely outside the client area.

If you set the **Alignment** property of your **StringFormat** object to **StringAlignment.Center**, the text string will be positioned so that the horizontal center corresponds with the $x$ coordinate passed to **DrawString**. If you set the **LineAlignment** property to **StringAlignment.Center**, the $y$ coordinate passed to **DrawString** indicates where the vertical center of the text string is positioned.

Here's how to use both effects to center text in the client area.

**HelloCenteredAlignment.cs**

```
//-----------------------------------------------------
// HelloCenteredAlignment.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
```
using System.Drawing;
using System.Windows.Forms;

class HelloCenteredAlignment: Form
{
    public static void Main()
    {
        Application.Run(new HelloCenteredAlignment());
    }

    public HelloCenteredAlignment()
    {
        Text = "Hello Centered Using String Alignment";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        StringFormat strfmt = new StringFormat();

        strfmt.Alignment = StringAlignment.Center;
        strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString("Hello, world!", Font, new SolidBrush(ForeColor),
                        ClientSize.Width / 2, ClientSize.Height / 2,
                        strfmt);
    }
}

And here it is:

I'm going to warn you about something that sounds pretty stupid, but I'm speaking from experience here. Often when I add a StringFormat definition to some existing code, I remember to do everything except include the object as the last argument to DrawString. Because DrawString doesn't require StringFormat, the program compiles just fine but the StringFormat doesn't seem to make any difference. You really need to include it in the DrawString call for it to work right!

**Measuring the String**

Another approach to centering text—a much more generalized approach to text positioning—doesn't require the `StringFormat` class but instead involves a method of the `Graphics` class, named `MeasureString`. `MeasureString` comes in seven versions, the simplest of which you call something like this:

```csharp
SizeF sizefText = grfx.MeasureString(str, font);
```

`MeasureString` returns a `SizeF` structure that indicates the width and height of the string in units of pixels (or, as you'll discover in Chapter 7, whatever units you prefer). `MeasureString` is easily the second most important method for displaying text—not as essential as `DrawString`, but right up there nonetheless. I'll have more to say about `MeasureString` in Chapter 9.

Imagine a displayed text string. Now imagine a rectangle drawn around that string. The `SizeF` structure returned from `MeasureString` is the width and height of that rectangle. For a particular font, regardless of the character string, the `Height` property of the `SizeF` structure is always the same. (Actually, the `Height` property is usually independent of the character string. If the string has embedded line feed characters, the `Height` property represents the height of multiple lines of text and hence will be an integral multiple of the `Height` value for a single line of text.)

The `Width` property of the `SizeF` structure depends on the characters that comprise the string. For all but fixed-pitch fonts, the width of the text string "i" is less than the width of "W", and `MeasureString` reflects that difference.

We'll be using `MeasureString` a lot in this book. For now, to center some text in the client area, you can subtract those `Width` and `Height` properties of the `SizeF` structure returned from `MeasureString` from the width and height of the client area. The two differences represent the total horizontal and vertical margin around the text. Divide each value by 2, and that's where to position the upper left corner of the string. Here's the complete code.

```csharp
HelloCenteredMeasured.cs
//----------------------------------------------------
// HelloCenteredMeasured.cs © 2001 by Charles Petzold
//----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloCenteredMeasured: Form
{
    public static void Main()
    {
        Application.Run(new HelloCenteredMeasured());
    }
    public HelloCenteredMeasured()
    {
        Text = "Hello Centered Using MeasureString";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        string str = "Hello, world!";
        SizeF sizefText = grfx.MeasureString(str, Font);
```
Text in a Rectangle

We've already looked at two versions of the DrawString method. There are six in total:

**Graphics Class DrawString Methods**

```csharp
void DrawString(string str, Font font, Brush brush, PointF ptf)
void DrawString(string str, Font font, Brush brush, float x, float y)
void DrawString(string str, Font font, Brush brush, RectangleF rectf)
void DrawString(string str, Font font, Brush brush, PointF ptf,
    StringFormat sf)
void DrawString(string str, Font font, Brush brush, float x, float y,
    StringFormat sf)
void DrawString(string str, Font font, Brush brush, RectangleF rectf,
    StringFormat sf)
```

As you can see, the first three arguments are always the same. The only differences are whether you specify coordinates using a PointF structure, two float values, or a RectangleF and whether you include a StringFormat argument.

Whether you use a PointF structure or two float values is a matter of personal preference. The two methods have identical functionality. Use whichever is currently convenient in your program.

But the RectangleF version is a little different. The DrawString method confines the text to the rectangle, and the optional StringFormat argument governs how the text is positioned within the rectangle. For example, if ClientRectangle is passed to the DrawString function and the Alignment and LineAlignment properties of StringFormat are both set to StringAlignment.Center, the text is centered within the client area, as the following program demonstrates.

**HelloCenteredRectangle.cs**

```csharp
//-----------------------------------------------------
// HelloCenteredRectangle.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloCenteredRectangle: Form
{
    public static void Main()
    {
        Application.Run(new HelloCenteredRectangle());
    }
    public HelloCenteredRectangle()
There's only one way to find out. Let's replace that little text string we've been using with something more substantial—the first paragraph of Mark Twain's *The Adventures of Huckleberry Finn* comes to mind as a reasonable example—and see what happens.

HuckleberryFinn.cs

```
//----------------------------------------------
// HuckleberryFinn.cs © 2001 by Charles Petzold
//----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HuckleberryFinn: Form
{
    public static void Main()
    {
        Application.Run(new HuckleberryFinn());
    }
    public HuckleberryFinn()
    {
        Text = "The Adventures of Huckleberry Finn";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        StringFormat strfmt = new StringFormat();

        strfmt.Alignment = StringAlignment.Center;
        strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString("Hello, world!", Font, new SolidBrush(ForeColor),
                        ClientRectangle, strfmt);
    }
}
```
You don't know about me, without you " +
"have read a book by the name of "The " +
"Adventures of Tom Sawyer," but that " +
"ain't no matter. That book was made by " +
"Mr. Mark Twain, and he told the truth, " +
"mainly. There was things which he " +
stretched, but mainly he told the truth. " +
"That is nothing. I never seen anybody " +
"but lied, one time or another, without " +
"it was Aunt Polly, or the widow, or " +
"maybe Mary. Aunt Polly\x2014Tom's Aunt " +
"Polly, she is\x2014and Mary, and the Widow " +
"Douglas, is all told about in that book" +
"\x2014which is mostly a true book; with " +
some stretchers, as I said before.".

And sure enough, DrawString nicely formats the text to fit within the client area:

What DrawString doesn't do (and we can hardly expect it to) is give us a couple scroll bars if the client area isn't big enough.

But that's OK. We'll find out how to do scroll bars in the next chapter.
Chapter 4: An Exercise in Text Output

Overview

The client area of a form is sometimes referred to as the program's *presentation area*. Here is where you present to the user your program's look and feel, its personality and idiosyncrasies, its virtues and character flaws. The appearance of a program's client area is, of course, highly dependent on what the program does. Some programs—particularly those that serve as front ends for distributed applications—may consist entirely of child window controls such as buttons and edit fields. Others may do all their own drawing, keyboard input, and mouse processing within the client area. And some programs—like the ones in this chapter—may simply display information.

Yet the simple display of information often requires some user interaction. This chapter discusses not only the techniques of formatting text into nice neat columns but also the ways of enabling and using those wonderful user-input devices known as *scroll bars*.

System Information

When I encounter a new operating system or a development environment like Windows Forms, I often write programs that explore the system itself. Writing programs that do nothing but display information (like the FormSize program in Chapter 3) gives me an opportunity to find out about the system while learning to code for it at the same time.

The *SystemInformation* class in the *System.Windows.Forms* namespace contains (at last count) 60 static read-only properties that reveal certain aspects of the particular computer on which your application is running and certain metrics the system uses to display items on the desktop and in your program. *SystemInformation* tells you the number of buttons on the user's mouse, the size of icons on the desktop, and the height of the form's caption bar. It also indicates whether the computer is connected to a network and the name of the user's domain. This information is returned in a variety of data types—*int*, *bool*, *string*, *Size*, *Rectangle*—and a couple enumerations.

My mission in this chapter is to create a program that displays this information in a manner that is convenient to peruse. Because you'll probably consult this program fairly often, doing a good job will be worth the extra effort.

Spacing Lines of Text

As you saw in Chapter 3, *DrawString* properly spaces multiple lines of text that are separated by line feed characters, and it also wraps text in a rectangle. What's usually more convenient in a program that displays multiple lines of text in columns, however, is to call *DrawString* for each line of each column separately. That means specifying a coordinate point in *DrawString* that indicates exactly where each text string goes.

In Chapter 3, I introduced the *MeasureString* method of the *Graphics* class. That method gives us a height of a character string. Although you can use this height for spacing successive lines of text, it's not quite suitable for that purpose. For performing line spacing that's consistent with the word-wrapping facility of *DrawString*, you should use a value that's a little different than the height returned from *MeasureString*. This subject is a bit confusing because the properties and methods that provide you with proper line-spacing values have names that seem to refer to the height of the font characters! The most generalized method for obtaining a line-spacing value is this *GetHeight* method of the *Font* class:

```csharp
float cySpace = font.GetHeight(grfx);
```

I use a variable name prefix of *cy* to mean a *count* in the *y* direction, that is, a height. In this statement, *font* is an object of type *Font* and *grfx* is an object of type *Graphics*. I refer to this as the most generalized method because the *Graphics* argument allows it to be used for both the video display and the printer. The method also takes into account any transforms that are in effect. (Transforms allow you to draw in units other than pixels, as I'll explain in Chapter 7.) Notice that the return value is a *float*. With some fonts, the value returned from *GetHeight* is the same as the height associated with *MeasureString*. For most fonts, however, *GetHeight* is somewhat smaller.
Another version of the *GetHeight* method gives you a line-spacing value that is suitable only for the video display and not the printer. You should use this method only when you’re drawing on the video display and when no transforms are in effect:

```csharp
float cySpace = font.GetHeight();
```

If you round that *float* value up to the next highest integer, you'll obtain the same value that's returned from the *Height* property of the *Font* class:

```csharp
int cySpace = font.Height;
```

If you're drawing on the video display in units of pixels (which is the default), the *Height* property of *Font* is probably the best choice.

When you use the *GetHeight* method or *Height* property with the default font associated with the form, you can just specify the form's *Font* property as the *Font* object:

```csharp
float cySpace = Font.GetHeight();
int cySpace = Font.Height;
```

Notice the uppercase *Font* in these statements to refer to the *Font* property. The *Form* class includes a protected read/write property named *FontHeight* (inherited from *Control*) that returns an *int* value consistent with the *Font.Height* property. Although in theory you can set this property, doing so doesn't result in the form's default font changing size.

### Property Values

Here's a first—and woefully incomplete—stab at writing a program to display *SystemInformation* properties.

#### SysInfoFirstTry.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoFirstTry: Form
{
    public static void Main()
    {
        Application.Run(new SysInfoFirstTry());
    }
    public SysInfoFirstTry()
    {
        Text = "System Information: First Try";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx  = pea.Graphics;
        Brush brush = new SolidBrush(ForeColor);
        int y = 0;
    }
}```
grfx.DrawString("ArrangeDirection: " +
    SystemInformation.ArrangeDirection,
    Font, brush, 0, y);
grfx.DrawString("ArrangeStartingPosition: " +
    SystemInformation.ArrangeStartingPosition,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("BootMode: " +
    SystemInformation.BootMode,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("Border3DSize: " +
    SystemInformation.Border3DSize,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("BorderSize: " +
    SystemInformation.BorderSize,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("CaptionButtonSize: " +
    SystemInformation.CaptionButtonSize,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("CaptionHeight: " +
    SystemInformation.CaptionHeight,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("ComputerName: " +
    SystemInformation.ComputerName,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("CursorSize: " +
    SystemInformation.CursorSize,
    Font, brush, 0, y += Font.Height);
grfx.DrawString("DbcsEnabled: " +
    SystemInformation.DbcsEnabled,
    Font, brush, 0, y += Font.Height);
}  

Well, I gave up after 10 items, not because I got tired of typing, but because I realized that this wasn't the best approach and that I'd probably need to retype the items in some other, more generalized format. As far as it goes, though, it's not bad for a first try:
Let's take a look at how this program works.

Each line of output is a single call to `DrawString`. The first argument is the text name of the property concatenated with the property value. The `SystemInformation` property is automatically converted into a string for the concatenation. What happens behind the scenes is a call to the `ToString` method implemented by the property's return value. In particular, notice that those properties returning `Size` values get nicely formatted to indicate the `Width` and `Height` values. The proper line spacing is handled within the `DrawString` call. Each `DrawString` call after the first has a `y` coordinate value of

\[
y += \text{Font.Height}
\]

thus placing it one line lower in the client area.

### Formatting into Columns

Other than its incompleteness, I think the most glaring problem with SysInfoFirstTry is the formatting. The output of a program like this would be easier to read if the property values were formatted into a second column. So let's tackle that problem before continuing onward.

Of the 10 properties that SysInfoFirstTry displays, the widest property name seems to be `ArrangeStartingPosition`. Before displaying any information, this program calls `MeasureString` with that string (plus a space so that the two columns won't touch).

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoColumns : Form
{
    public static void Main()
    {
        Application.Run(new SysInfoColumns());
    }
    public SysInfoColumns()
    {
```
Text = "System Information: Columns";
BackColor = SystemColors.Window;
ForeColor = SystemColors.WindowText;
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Brush brush = new SolidBrush(ForeColor);
    SizeF sizef;
    float cxCol, y = 0;
    int cySpace;

    sizef = grfx.MeasureString("ArrangeStartingPosition ", Font);
    cxCol = sizef.Width;
    cySpace = Font.Height;

    grfx.DrawString("ArrangeDirection", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.ArrangeDirection.ToString(),
                    Font, brush, cxCol, y);
    y += cySpace;

    grfx.DrawString("ArrangeStartingPosition", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.ArrangeStartingPosition.ToString(),
                    Font, brush, cxCol, y);
    y += cySpace;

    grfx.DrawString("BootMode", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.BootMode.ToString(),
                    Font, brush, cxCol, y);
    y += cySpace;

    grfx.DrawString("Border3DSize", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.Border3DSize.ToString(),
                    Font, brush, cxCol, y);
    y += cySpace;

    grfx.DrawString("BorderSize", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.BorderSize.ToString(),
                    Font, brush, cxCol, y);
    y += cySpace;

    grfx.DrawString("CaptionButtonSize", Font, brush, 0, y);
    grfx.DrawString(SystemInformation.CaptionButtonSize.ToString(),
                    Font, brush, cxCol, y);
The program saves the width of the string in the variable cxCol and uses that to position the second column. The program also saves the Height property of the form's Font object in a variable named cySpace and uses that to space successive lines of text. The coordinate positions passed to the DrawString method are float values.

Now each line of output requires two calls to DrawString, the first displaying the property name and the second displaying the property value. All but one of these property values now require explicit ToString calls to convert the values to strings. Here's what it looks like:

```
Everything Is an Object
```
In a program like SysInfoColumns, the code that displays the lines of text should probably be in a `for` loop. The actual information should probably be stored in an array of some sort, and perhaps isolated from the actual text-output code so that it could be used in other programs. In a C or C++ program, for example, you might put the information in an array of structures in a header file. However, there are no header files in C#. Instead, remember the magic rule for C#: *Everything is an object*—or at least a class with static methods and properties.

Here's one possible implementation of a class that contains the text strings we want to display and provides some information about them.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoStrings
{
    public static string[] Labels
    {
        get
        {
            return new string[]
            {
                "ArrangeDirection",
                "ArrangeStartingPosition",
                "BootMode",
                "Border3DSize",
                "BorderSize",
                "CaptionButtonSize",
                "CaptionHeight",
                "ComputerName",
                "CursorSize",
                "DbcsEnabled",
                "DebugOS",
                "DoubleClickSize",
                "DoubleClickTime",
                "DragFullWindows",
                "DragSize",
                "FixedFrameBorderSize",
                "FrameBorderSize",
                "HighContrast",
                "HorizontalScrollBarArrowWidth",
                "HorizontalScrollBarHeight",
                "HorizontalScrollBarThumbWidth",
                "IconSize",
                "IconSpacingSize",
            }
        }
    }
}
```

"KanjiWindowHeight",
"MaxWindowTrackSize",
"MenuButtonSize",
"MenuCheckSize",
"MenuFont",
"MenuHeight",
"MidEastEnabled",
"MinimizedWindowSize",
"MinimizedWindowSpacingSize",
"MinimumWindowSize",
"MinWindowTrackSize",
"MonitorCount",
"MonitorsSameDisplayFormat",
"MouseButtons",
"MouseButtonsSwapped",
"MousePresent",
"MouseWheelPresent",
"MouseWheelScrollLines",
"NativeMouseWheelSupport",
"Network",
"PenWindows",
"PrimaryMonitorMaximizedWindowSize",
"PrimaryMonitorSize",
"RightAlignedMenus",
"Secure",
"ShowSounds",
"SmallIconSize",
"ToolWindowCaptionButtonSize",
"ToolWindowCaptionHeight",
"UserDomainName",
"UserInteractive",
"UserName",
"VerticalScrollBarArrowHeight",
"VerticalScrollBarThumbHeight",
"VerticalScrollBarWidth",
"VirtualScreen",
"WorkingArea",

};

public static string[] Values
{
    get
    {
        return new string[]
SystemInformation.ArrangeDirection.ToString(),
SystemInformation.ArrangeStartingPosition.ToString(),
SystemInformation.BootMode.ToString(),
SystemInformation.Border3DSize.ToString(),
SystemInformation.BorderSize.ToString(),
SystemInformation.CaptionButtonSize.ToString(),
SystemInformation.CaptionHeight.ToString(),
SystemInformation.ComputerName,
SystemInformation.CursorSize.ToString(),
SystemInformation.DbcsEnabled.ToString(),
SystemInformation.DebugOS.ToString(),
SystemInformation.DoubleClickSize.ToString(),
SystemInformation.DoubleClickTime.ToString(),
SystemInformation.DragFullWindows.ToString(),
SystemInformation.DragSize.ToString(),
SystemInformation.FixedFrameBorderSize.ToString(),
SystemInformation.FrameBorderSize.ToString(),
SystemInformation.HighContrast.ToString(),
SystemInformation.HorizontalScrollBarArrowWidth.ToString(),
SystemInformation.HorizontalScrollBarHeight.ToString(),
SystemInformation.HorizontalScrollBarThumbWidth.ToString(),
SystemInformation.IconSize.ToString(),
SystemInformation.IconSpacingSize.ToString(),
SystemInformation.KanjiWindowHeight.ToString(),
SystemInformation.MaxWindowTrackSize.ToString(),
SystemInformation.MenuButtonSize.ToString(),
SystemInformation.MenuCheckSize.ToString(),
SystemInformation.MenuFont.ToString(),
SystemInformation.MenuHeight.ToString(),
SystemInformation.MidEastEnabled.ToString(),
SystemInformation.MinimizedWindowSize.ToString(),
SystemInformation.MinimizedWindowSpacingSize.ToString(),
SystemInformation.MinimumWindowSize.ToString(),
SystemInformation.MinWindowTrackSize.ToString(),
SystemInformation.MonitorCount.ToString(),
SystemInformation.MonitorsSameDisplayFormat.ToString(),
SystemInformation.MouseButtons.ToString(),
SystemInformation.MouseButtonsSwapped.ToString(),
SystemInformation.MousePresent.ToString(),
SystemInformation.MouseWheelPresent.ToString(),
SystemInformation.MouseWheelScrollLines.ToString(),
SystemInformation.NativeMouseWheelSupport.ToString(),
SystemInformation.Network.ToString(),
SystemInformation.PenWindows.ToString(),
SystemInformation.PrimaryMonitorMaximizedWindowSize.ToString(),
SystemInformation.PrimaryMonitorSize.ToString(),
SystemInformation.RightAlignedMenus.ToString(),
SystemInformation.Secure.ToString(),
SystemInformation.ShowSounds.ToString(),
SystemInformation.SmallIconSize.ToString(),
SystemInformation.ToolWindowCaptionButtonSize.ToString(),
SystemInformation.ToolWindowCaptionHeight.ToString(),
SystemInformation.UserDomainName,
SystemInformation.UserInteractive.ToString(),
SystemInformation.UserName,
SystemInformation.VerticalScrollBarArrowHeight.ToString(),
SystemInformation.VerticalScrollBarThumbHeight.ToString(),
SystemInformation.VerticalScrollBarWidth.ToString(),
SystemInformation.VirtualScreen.ToString(),
SystemInformation.WorkingArea.ToString(),
);

public static int Count
{
    get
    {
        return Labels.Length;
    }
}

public static float MaxLabelWidth(Graphics grfx, Font font)
{
    return MaxWidth(Labels, grfx, font);
}

public static float MaxValueWidth(Graphics grfx, Font font)
{
    return MaxWidth(Values, grfx, font);
}

static float MaxWidth(string[] astr, Graphics grfx, Font font)
{
    float fMax = 0;

    foreach (string str in astr)
    {
        fMax = Math.Max(fMax, grfx.MeasureString(str, font).Width);
    }

    return fMax;
}
This class has three read-only static properties. The first, *Labels*, returns an array of strings that are the names of the *SystemInformation* properties.

The second property is named *Values*, and it also returns an array of character strings, each one corresponding to an element of the *Labels* array. However, in reality, the *Values* array consists of a series of expressions involving *SystemInformation* properties, each of which evaluates to a string. Each expression is evaluated at the time the property is obtained. You'll recognize the use of the *ToString* method to convert each item to a string.

The third property returns the number of items in the *Labels* array, which should (unless I messed up) also be the number of items in the *Values* array.

In addition, *SysInfoStrings* has two public methods: *MaxLabelWidth* and *MaxValueWidth*. These two methods require arguments of a *Graphics* object and a *Font* object and simply return the largest width reported by *MeasureString* for the two arrays. They both rely on a private method named *MaxWidth*. That method makes use of the static *Math.Max* method to obtain the maximum of two values. (The *Math* class in the *System* namespace is a collection of static methods that implement various mathematics functions. Appendix B is devoted to the *Math* class and other aspects of working with numbers.)

**Listing the System Information**

I created the SysInfoStrings.cs file in a project named SysInfoList, which also includes the SysInfoList.cs file shown here. This program displays the system information items provided by the *SysInfoStrings* class.

**SysInfoList.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoList: Form
{
    readonly float cxCol;
    readonly int   cySpace;

    public static void Main()
    {
        Application.Run(new SysInfoList());
    }
    public SysInfoList()
    {
        Text = "System Information: List";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        Graphics grfx = CreateGraphics();
        SizeF    sizef = grfx.MeasureString( " ", Font);
        cxCol    = sizef.Width + SysInfoStrings.MaxLengthWidth(grfx, Font);
        grfx.Dispose();
        cySpace = Font.Height;
    }
}
```
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Brush     brush = new SolidBrush(ForeColor);
    int      iCount = SysInfoStrings.Count;
    string[] astrLabels = SysInfoStrings.Labels;
    string[] astrValues = SysInfoStrings.Values;

    for (int i = 0; i < iCount; i++)
    {
        grfx.DrawString(astrLabels[i], Font, brush,
                        0, i * cySpace);
        grfx.DrawString(astrValues[i], Font, brush,
                        cxCol, i * cySpace);
    }
}

This program attempts to be somewhat efficient by calculating the cxCol and cySpace values only once during the form's constructor, saving them as fields that the OnPaint method uses later. However, this calculation requires a call to MeasureString, and MeasureString requires a Graphics object. How do we get such an object outside of a Paint event or an OnPaint method call?

The Control class implements (and the Form class inherits) a CreateGraphics method that lets you obtain a Graphics object at any time:

**Control Methods (selection)**

Graphics CreateGraphics()

You can use this Graphics object to obtain information—as the SysInfoList program does—or to draw on the program's client area. (However, don't bother drawing during the constructor because the form isn't even displayed at that time.)

You must dispose of a Graphics object obtained from CreateGraphics before returning control from the constructor or the event in which it is created. You do this by calling the Dispose method:

**Graphics Methods (selection)**

void Dispose()

After SysInfoList obtains a Graphics object from CreateGraphics, it calls MeasureString to obtain the width of a single space character. It adds this width to the MaxLabelWidth value returned from the SysInfoStrings class and saves that result as cxCol. A simple for loop displays the pair of strings during the OnPaint method.

And we're definitely making progress:
However, depending on certain settings you've made regarding your video display resolution (an issue I'll discuss in Chapter 9), you may or may not be able to resize the form to be large enough to view all 60 values. If you can't, you'll find this program very frustrating to use. There's no way to bring the later values into view.

And even if there were only 20 or 25 values, you might still face a problem. Perhaps the worst mistake you can make in Windows programming is to assume that a certain amount of text is visible on a particular user's machine. Users—particularly those whose eyesight isn't too good—can set large font sizes and thus reduce the amount of text that is visible on the screen. Your Windows programs should be usable at just about any screen resolution and font size.

We need to display more text than can fit on the screen, but that's nothing a scroll bar can't fix.

**Windows Forms and Scroll Bars**

Scroll bars are an important part of any graphical environment. For the user, they are easy to use and provide excellent visual feedback. You can use scroll bars whenever you need to display anything—text, graphics, a spreadsheet, database records, pictures, Web pages—that requires more space than is available in the program's client area.

Scroll bars are oriented either vertically, for up-and-down movement, or horizontally, for back-and-forth movement. Clicking the arrows at either end of a scroll bar causes the document to scroll by a small amount—generally a line of text for a vertical scroll bar. Clicking the area between the arrows causes the document to scroll by a larger amount. A scroll box (also sometimes called the scroll bar thumb) travels the length of the scroll bar to indicate the approximate location of the material shown in the client area in relation to the entire content. You can drag the scroll box with the mouse to move to a particular location within the content. A relatively recent innovation in scroll bars makes the size of the scroll box variable to indicate the relative proportion of the content currently displayed in the client area.

You can add scroll bars to a form in one of two ways. In the first approach, you create controls of type VScrollBar and HScrollBar and position them anywhere in the client area. These scroll bar controls have settable properties that affect the appearance and functionality of the scroll bars. A form is notified when the user manipulates a scroll bar control through events. I'll be putting scroll bar controls to work in Chapter 12.

The second approach to adding scroll bars to a form is easier than creating scroll bar controls. This approach, often called the auto-scroll approach, is the one I'll be demonstrating in this chapter.

The auto-scroll facility is primarily intended for programs that put controls (such as buttons and text boxes) on the form's client area. The program enables auto-scroll by setting the AutoScroll property of the form to true. If the client area is too small to allow all the controls to be visible at once, scroll bars appear (as if by magic) that allow the user to bring the other controls into view.

It's also possible to enable auto-scroll without using any controls. I'll show you both approaches, and you can decide which you like best.

**Scrolling a Panel Control**

The .NET Framework has lots of interesting controls, ranging from buttons, list boxes, and text boxes to calendars, tree views, and data grids. The panel control, however, is not one of these interesting controls. It has no visual appearance to speak of and not much of a user interface. Panels
generally used for architectural purposes to group other controls against a background. Panels are also useful when you need a control but don't want it to do very much.

What I've done in the following SysInfoPanel program is to create a panel control that is the size of the information I want to display—that is, a panel control large enough to display all 60 lines of system-information text. I put that control on the client area of the form and let auto-scrolling do the rest.

SysInfoPanel.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoPanel : Form
{
    readonly float cxCol;
    readonly int cySpace;

    public static void Main()
    {
        Application.Run(new SysInfoPanel());
    }

    public SysInfoPanel()
    {
        Text = "System Information: Panel";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        AutoScroll = true;

        Graphics grfx = CreateGraphics();
        SizeF sizef = grfx.MeasureString(" ", Font);
        cxCol = sizef.Width + SysInfoStrings.MaxValueWidth(grfx, Font);
        cySpace = Font.Height;

        // Create a panel.

        Panel panel = new Panel();
        panel.Parent = this;
        panel.Paint += new PaintEventHandler(PanelOnPaint);
        panel.Location = Point.Empty;
        panel.Size = new Size((int)Math.Ceiling(cxCol +
                                SysInfoStrings.MaxValueWidth(grfx, Font)),
                               (int)Math.Ceiling(cySpace * SysInfoStrings.Count));
```
grfx.Dispose();
}

void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Brush brush = new SolidBrush(ForeColor);
    int iCount = SysInfoStrings.Count;
    string[] astrLabels = SysInfoStrings.Labels;
    string[] astrValues = SysInfoStrings.Values;

    for (int i = 0; i < iCount; i++)
    {
        grfx.DrawString(astrLabels[i], Font, brush, 0, i * cySpace);
        grfx.DrawString(astrValues[i], Font, brush, cxCol, i * cySpace);
    }
}

This program also requires the SysInfoStrings.cs file. A good way to share files among projects in Visual C# .NET is to use the Add Existing Item menu item. (You'll find this entry on the Project menu; you can also select it by right-clicking the project name in Solution Explorer and selecting Add.) You select the existing file you need in the project, and instead of pressing the Open button, you click the arrow next to Open and select Link File. Doing this avoids making a copy of the file and also prevents problems that result when you change one copy but not the other.

Let's look at the SysInfoPanel constructor. To enable the auto-scroll facility, you must set the AutoScroll property of the form to true. That's the easy part. Next the program calculates cxCol and cySpace exactly as SysInfoList did. But before disposing of the Graphics object, the program proceeds to create the panel
Panel panel = new Panel();
I've given this panel the name panel. I want this panel to be located on the surface of the form's client area. The surface on which a control is located is called the control's parent. All controls must have a parent. The statement
panel.Parent = this;
assigns the program's form to be the parent of the panel. The keyword this is used in a method or property to refer to the current instance of the object; here, this refers to the particular Form object that I created. It's the same as the value passed to Application.Run in Main.

I want to be able to draw on this panel. To do that, I assign an event handler to the panel's Paint event:
panel.Paint += new PaintEventHandler(PanelOnPaint);
I used similar code in Chapter 2 when I installed Paint event handlers for forms that were instantiated from Form rather than inherited from it. The PanelOnPaint method is in the SysInfoPanel class.

The panel must have a location relative to its parent. The Location property of the Panel class indicates where the upper left corner of the panel will be positioned. In this program, I want the panel's upper left corner to be at the point (0, 0) in the client area, which I set with this statement:
panel.Location = Point.Empty;
This statement isn't strictly needed because the location of controls is at point (0, 0) by default.

For this program to work correctly, the panel's size must be set to the dimensions of the output you want to display:

```csharp
panel.Size = new Size(
    (int) Math.Ceiling(cxCol + SysInfoStrings.MaxValueWidth(grfx, Font)),
    (int) Math.Ceiling(cySpace * SysInfoStrings.Count));
```

The `cxCol` variable has already been set to the width of the first column plus a space. The call to the `MaxValueWidth` method of `SysInfoStrings` retrieves the maximum width of the `SystemInformation` property values. The height of the panel is set equal to the line-spacing value times the number of lines. I use `Math.Ceiling` to round each value up to the next highest integer. (An alternative is to make a `SizeF` object based on the floating-point width and height, and then use the static `Size.Ceiling` method to convert it to a `Size` object.)

And that's that. The `PanelOnPaint` method displays the information on the surface of the panel using exactly the same code as the `OnPaint` method in the `SysInfoList` class.

But in this program, whenever the panel is wider than the client area, a horizontal scroll bar will automatically appear. If the panel is higher than the height of the client area, a vertical scroll bar will appear at the right of the client area. This happens dynamically: as you change the size of the client area, the scroll bars will disappear and reappear as needed. The scroll boxes are also dynamically sized to reflect the proportion of the content that is visible. For example, the height of the vertical scroll box is based on the ratio of the client area height to the height of the panel:

![System Information: Panel](image)

Because controls generally adopt the background colors of their parents and because panels are such bland controls to begin with, it's hard to see that there's really another control here. To give yourself a better idea of what's going on in this program, you may want to explicitly give the panel a different background color:

```csharp
panel.BackColor = Color.Honeydew;
```

When you then make the client area of the program larger than the panel—in which case, the scroll bars disappear—you can see the honeydew panel against the (probably white) background of the form. Another way to see the panel is to set the `AutoSizeMargin` property of the form in the constructor:

```csharp
AutoScrollMargin = new Size(10, 10);
```

You'll see a 10-pixel area on the right side of the panel when you scroll all the way to the right and on the bottom of the panel when you scroll all the way down. That's the background of the form's client area.

I mentioned earlier that a more general-purpose approach to scrolling involves the use of scroll bar controls. Scroll bars created as controls have properties named `Minimum` and `Maximum` that define the numeric values associated with the extreme positions of the scroll box and thus the range of values that the scroll bar can assume. When using the auto-scroll facility, however, you don't have access to these settings. The range is implied by the difference between the width and height of the client area less the width and height of the area occupied by the controls (or in our case, the single `Panel` control) plus the `AutoSizeMargin` less the width and height of the client area.
Scroll bars created as controls generate an event named `Scroll` when the user manipulates the scroll bar. There is no such event associated with auto-scroll—at least not that an application has access to.

Although the SysInfoPanel program isn't responding directly to `Scroll` events, it's definitely responding to `Paint` events from the panel. When a program paints on a control, it's really painting only on the visible area of the control. Every time the user scrolls, the panel generates a `Paint` event because some previously unseen area has been pulled into view.

**The Heritage of `ScrollableControl`**

What's going on behind the scenes? As I explained in Chapter 2, among the many classes that `Form` descends from is the `ScrollableControl` class, and that's where auto-scroll is implemented. We've already encountered two of the following six properties of `ScrollableControl` that are also inherited by `Form`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td><code>AutoScroll</code></td>
<td>get/set</td>
<td>Enables auto-scroll</td>
</tr>
<tr>
<td>bool</td>
<td><code>HScroll</code></td>
<td>get/set</td>
<td>Indicates the existence of horizontal scroll</td>
</tr>
<tr>
<td>bool</td>
<td><code>VScroll</code></td>
<td>get/set</td>
<td>Indicates the existence of vertical scroll</td>
</tr>
<tr>
<td>Size</td>
<td><code>AutoScrollMargin</code></td>
<td>get/set</td>
<td>Sets the margin around right and bottom of controls</td>
</tr>
<tr>
<td>Size</td>
<td><code>AutoScrollMinSize</code></td>
<td>get/set</td>
<td>Defines the minimum scrolling area</td>
</tr>
<tr>
<td>Point</td>
<td><code>AutoScrollPosition</code></td>
<td>get/set</td>
<td>Indicates the scroll bar position</td>
</tr>
</tbody>
</table>

You can determine whether a particular scroll bar is currently visible by using the `HScroll` and `VScroll` properties. (Supposedly, you can also use these properties to hide a scroll bar that would normally be visible, but that facility doesn't seem to work very well.) I'll be discussing `AutoScrollMinSize` in more detail shortly.

`AutoScrollPosition` provides the current scrolling position in negative coordinates. In the SysInfoPanel program, the value of `AutoScrollPosition` is the same as the value of the panel's `Location` property. However, there's an inconsistency in the `get` and `set` accessors of `AutoScrollPosition`. When you read the property, the coordinates are always less than or equal to 0. When you set `AutoScrollPosition`, however, the coordinates must be positive. I'll have an example of this anomaly in the SysInfoKeyboard program in Chapter 6 when I add a keyboard interface to the program.

The `ScrollableControl` class obviously has access to the normal properties and events of the scroll bars; the class is hiding these items from you in order to provide a higher-level interface. As you manipulate the scroll bar in the SysInfoPanel program, code implemented in `ScrollableControl` is obviously changing the `Location` property of the panel control to negative values. (It's easy to confirm this change by adding an event handler for the panel's `Move` event.) Negative `Location` values mean that the upper left corner of the panel is being positioned above and to the left of the upper left corner of the client area. That's why the contents of the panel seem to move around within the form.

We'll explore auto-scroll more in later chapters as we begin creating more controls. Now let's see if we can persuade auto-scroll to work without creating any child controls at all.

**Scrolling Without Controls**

The key to enabling auto-scroll without creating child controls is to set the `AutoScrollMinSize` property to something other than the default (0, 0). Normally, the scrolling area is based on the locations and sizes of controls on the client area. However, `AutoScrollMinSize` sets a minimum scrolling area regardless of the presence of any controls. Of course, you must also set the `AutoScroll` property to `true`.

Typically, you set `AutoScrollMinSize` to the dimensions necessary to display all the program's output. In the system-information programs, `AutoScrollMinSize` should be set to a size sufficient to
encompass the full width and height of all 60 lines of information. That's the same size as the panel in the SysInfoPanel program.

The SysInfoScroll program is virtually identical in functionality to SysInfoPanel but enables auto-scroll without any child controls.

SysInfoScroll.cs
//--------------------------------------------
// SysInfoScroll.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoScroll: Form
{
    readonly float cxCol;
    readonly int cySpace;

    public static void Main()
    {
        Application.Run(new SysInfoScroll());
    }
    public SysInfoScroll()
    {
        Text = "System Information: Scroll";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        Graphics grfx = CreateGraphics();
        SizeF sizef = grfx.MeasureString(" ", Font);
        cxCol = sizef.Width + SysInfoStrings(MaxLabelWidth(grfx, Font));
        cySpace = Font.Height;

        // Set auto-scroll properties.
        AutoScroll = true;
        AutoScrollMinSize = new Size((int)Math.Ceiling(cxCol +
            SysInfoStrings.MaxValueWidth(grfx, Font)),
            (int)Math.Ceiling(cySpace * SysInfoStrings.Count));

        grfx.Dispose();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
    
}
```csharp
Graphics grfx = pea.Graphics;
Brush brush = new SolidBrush(ForeColor);
int iCount = SysInfoStrings.Count;
string[] astrLabels = SysInfoStrings.Labels;
string[] astrValues = SysInfoStrings.Values;
Point pt = AutoScrollPosition;

for (int i = 0; i < iCount; i++)
{
    grfx.DrawString(astrLabels[i], Font, brush, pt.X, pt.Y + i * cySpace);
    grfx.DrawString(astrValues[i], Font, brush, pt.X + cxCol, pt.Y + i * cySpace);
}
```

SysInfoPanel put a panel control on its client area. In a program such as SysInfoScroll, you might wonder whether the scroll bars are also located on top of the client area. They are not! The client area is actually made smaller to accommodate the scroll bars. Sometimes the width of the client area is just a little larger than `AutoScrollMinSize.Width`. If a vertical scroll bar is required, however, the width of the client area must be narrowed by the width of the scroll bar; that change could then make the client area width smaller than `AutoScrollMinSize.Width` and thus also require a horizontal scroll bar.

By setting `AutoScrollMinSize`, you are not defining something akin to a virtual drawing area. Regardless of how large you make `AutoScrollMinSize`, when you handle the `OnPaint` method, you're still drawing within the confines of the physical client area. Indeed, that client area is probably even smaller than usual because of the presence of the scroll bars.

In SysInfoPanel, any manipulation of the scroll bars resulted in the uncovering of previously unseen areas of the panel control and thus generated a `Paint` event. That program did all its drawing on a panel control that was large enough for all the program's output. Scrolling relocated the panel relative to the program's client area. But the coordinates of the output on the panel remained the same. For example, the second column of the third row of text output was always at the location `(cxCol, 2 * cySpace)`.

SysInfoScroll responds to changes in the scroll bars by overriding the form's `OnPaint` method. However, this program is drawing directly on its client area and not on some control that's being shifted around. The client area isn't large enough for the program's output, and the `Graphics` object obtained during the `OnPaint` method knows nothing about auto-scroll.

What this means is that the `OnPaint` method of the SysInfoScroll program (or any program that implements auto-scroll and draws directly on its client area) must adjust the coordinates of any drawing function it calls based on the `AutoScrollPosition` property. As you can see, the `OnPaint` method in this program gets `AutoScrollPosition`, saves it in a `Point` variable named `pt`, and adds the values to the coordinates in the `DrawString` calls. Keep in mind that the coordinates returned by `AutoScrollPosition` are negative. If you've scrolled down 30 pixels (for example), the first `DrawString` call for the first line of text uses the coordinates `(0, -30)`, which is above the client area and not visible.

This method of repainting the client area may start to sound inefficient: The program is drawing 60 lines of text every time it needs to repaint, yet usually only a small fraction of those calls result in something being painted on the client area. I'll take on the efficiency issue later in this chapter.

**Actual Numbers**
Let's pause for a moment and try to get a better feel for what's going on by looking at actual numbers. Just keep in mind that some of these numbers are based on my system settings and may not be exactly the same numbers you're seeing. (In particular, my video display settings include Large Fonts. This setting affects some of the items I'll be discussing.)

Suppose your program needs a client area of 400 pixels wide by 1600 pixels high. Here's how you set AutoScroll and AutoScrollMinSize in the form's constructor:

```csharp
AutoScroll = true;
AutoScrollMinSize = new Size(400, 1600);
```

My experience is that forms are created with a default size of 300 by 300 pixels. How large is the client area in that case? Well, we now have two programs that let you scroll through the SystemInformation properties, so you can figure out how large the client area is. I see a SystemInformation.CaptionHeight value of 24. That's the height of the caption bar. The width of the normal sizing border is stored in SystemInformation.FrameBorderSize. I'm seeing 4 pixels for that, and remember that's 4 pixels on all four sides. So you can calculate the client area width as 300 minus two 4's, or 292. The height of the client area should be 300 minus two 4's minus 24, or 268. (If you don't trust my math skills, you can verify these values by using the form's ClientSize property.)

Because the client area height of 268 is less than 1600, the program needs a vertical scroll bar. I'm seeing a value of SystemInformation.VerticalScrollBarWidth of 20 pixels. Thus, the client area width is reduced to 292 minus 20, or 272 pixels.

That width of 272 is less than 400, so the program needs a horizontal scroll bar as well. The value of SystemInformation.HorizontalScrollBarHeight is also 20 pixels, thus reducing the height of the client area to 268 minus 20, or 248 pixels.

The vertical scroll bar is probably set to have a range of values from 0 through 1352, which is the required height of 1600 minus the actual height of 248. The horizontal scroll bar is probably set to have a range of values from 0 through 128 (which is 400 minus 272).

The code implemented in ScrollableControl responds to the user clicking the scroll bar or dragging the scroll box by performing two actions: changing the value of AutoScrollPosition and scrolling the contents of the client area. AutoScrollPosition is initially set to (0, 0). As the user moves the horizontal scroll bar, the X property varies between 0 and –128 and the Y property varies from 0 through –1352.

The scrolling of the client area requires the system to copy the contents from one location on the client area to another. The Win32 API includes functions named ScrollWindow, ScrollWindowEx, and ScrollDC that let programs scroll their client areas. Although these functions are not exposed in the Windows Forms class library, it's obvious that ScrollableControl is using one of them.

When code implemented in the ScrollableControl class scrolls the client area, it can scroll only what's currently displayed on the screen. Scrolling generally "uncovers" a rectangle in the client area, making that portion of the client invalid. This invalidation generates a call to the OnPaint method.

So when you're scrolling the SysInfoPanel or SysInfoScroll program, the OnPaint method really needs to refresh only a small rectangular subset of the client area. It hardly seems rational that these programs process the OnPaint call by obtaining and displaying every single line of information.

Let's take care of that problem in two steps.
LISPA ch4 Keeping it green
Don't Be a Pig

Users have a name for a program that isn't as fast as it could be. They say, "This program is a real pig." It's not nice, but it's a fact of life.

I've already made the system-information program somewhat more efficient by calling the methods in SysInfoStrings only when the program begins execution and when any of the SystemInformation items change. The program no longer makes three calls to SysInfoStrings every time it gets a call to OnPaint.

However, OnPaint is still displaying all 60 lines—and calling DrawString 120 times—every time any part of the client window is invalidated. On most people's machines, not all 60 lines will even be visible. Moreover, as I mentioned earlier, vertical scrolling usually uncovers only a line or two of text; in those cases, OnPaint really needs to redraw only a line or two.

To some extent, Windows itself provides some built-in efficiency. The Graphics object you obtain during the OnPaint method can paint only on the invalid region of the client area. Something called a clipping region, which encompasses only the invalid region and doesn't let you draw outside it, is involved. You saw an example of repainting only the invalid region in the RandomClear program in Chapter 3. The fact remains, however, that you're still making 120 DrawString calls, and you're still requiring Windows to check whether a particular DrawString call will or will not fall within the clipping region.

Fortunately, the ClipRectangle property of the PaintEventArgs class is there to help. The ClipRectangle property is the smallest rectangle in client area coordinates that encompasses the invalid region. (As the RandomClear program demonstrated, the invalid region need not be rectangular.) For personal experimentation, you might insert the line

```csharp
Console.WriteLine(pea.ClipRectangle);
```

in an OnPaint method and play with scrolling and partially covering and then uncovering the form with other programs.

The SysInfoEfficient program inherits from SysInfoUpdate and overrides the OnPaint method in that class with a more efficient version. A couple of fairly simple calculations based on the AutoScrollPosition property of the form and the ClipRectangle property of PaintEventArgs derive line index values named iFirst and iLast that are then used in the for loop to display the minimum number of lines of text required to update the client area.

```csharp
// SysInfoEfficient.cs
using System;
using System.Drawing;
using System.Windows.Forms;
class SysInfoEfficient : SysInfoUpdate
{
    public new static void Main()
    {
        Application.Run(new SysInfoEfficient());
    }
    public SysInfoEfficient()
    {
        Text = "System Information: Efficient";
    }
}
```
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx  = pea.Graphics;
    Brush brush    = new SolidBrush(ForeColor);
    Point pt      = AutoScrollPosition;

    int iFirst = (int)((pea.ClipRectangle.Top    - pt.Y) / cySpace);
    int iLast  = (int)((pea.ClipRectangle.Bottom - pt.Y) / cySpace);

    iLast = Math.Min(iCount - 1, iLast);

    for (int i = iFirst; i <= iLast; i++)
    {
        grfx.DrawString(astrLabels[i], Font, brush,
                         pt.X, pt.Y + i * cySpace);

        grfx.DrawString(astrValues[i], Font, brush,
                         pt.X + cxCol, pt.Y + i * cySpace);
    }
}

Just prior to the for loop, the statement involving Math.Min prevents iLast from exceeding the number of items to be displayed. This limit can be exceeded only if the window is taller than the size necessary to display all the items.

Reflecting on the Future

While the .NET Framework might appear to be the epitome of perfection today, there’s still a possibility, however slim, that in some distant year a misguided Microsoft developer might feel compelled to add one or two additional properties to the SystemInformation class. In that case, my SysInfoStrings class would have to be updated to include those additional properties, and all the various programs in this chapter would also have to be recompiled to include the new version.

Might it be possible, however, to write a program that automatically includes all current SystemInformation properties implemented in the class, even those that didn’t exist when the program was written?

Yes, it is possible, and to understand how to do it, let’s think about where the SystemInformation code actually exists. According to the documentation for the class, it’s in the file System.Windows.Forms.dll. When one of the programs in this chapter is run, the operating system links it with System.Windows.Forms.dll so that the program can make calls to the SystemInformation class.

But the DLL isn’t just a bunch of code. It exists with binary metadata that describes in detail the classes implemented in the file and all the fields, properties, methods, and events in these classes. In fact, the C# compiler uses this information to compile programs (that’s why you need to set the Reference files), and the reference documentation of the .NET Framework is derived from this metadata.

So it makes sense that a program might be able to access this metadata at runtime, find out about the .NET classes dynamically, and even execute some methods and properties in them. This process is called reflection, and it’s a concept borrowed from Java. Reflection would normally be considered an advanced topic, but it’s just so perfect for this application that I can’t resist.
The first step is to rewrite the `SysInfoStrings` class so that it uses reflection to obtain the property names and the actual properties.

**SysInfoReflectionStrings.cs**

```csharp
//-------------------------------------------------------
// SysInfoReflectionStrings.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;

class SysInfoReflectionStrings
{
    // Fields
    static bool bValidInfo = false;
    static int iCount;
    static string[] astrLabels;
    static string[] astrValues;

    // Constructor
    static SysInfoReflectionStrings()
    {
        SystemEvents.UserPreferenceChanged +=
            new UserPreferenceChangedEventHandler(UserPreferenceChanged);

        SystemEvents.DisplaySettingsChanged +=
            new EventHandler(DisplaySettingsChanged);
    }

    // Properties
    public static string[] Labels
    {
        get
        {
            GetSysInfo();
            return astrLabels;
        }
    }

    public static string[] Values
    {
        get
        {
            GetSysInfo();
            return astrValues;
        }
    }

    // Methods
    static void GetSysInfo()
    {
        // Implementation
    }

    static void UserPreferenceChanged(object sender, UserPreferenceChangedEventArgs e)
    {
        // Implementation
    }

    static void DisplaySettingsChanged(object sender, EventArgs e)
    {
        // Implementation
    }
}
```
public static int Count
{
    get
    {
        GetSysInfo();
        return iCount;
    }
}

// Event handlers
static void UserPreferenceChanged(object obj, UserPreferenceChangedEventArgs ea)
{
    bValidInfo = false;
}
static void DisplaySettingsChanged(object obj, EventArgs ea)
{
    bValidInfo = false;
}

// Methods
static void GetSysInfo()
{
    if(bValidInfo)
        return;

    // Get property information for SystemInformation class.
    Type type = typeof(SystemInformation);
    PropertyInfo[] apropinfo = type.GetProperties();

    // Count the number of static readable properties.
    iCount = 0;
    foreach (PropertyInfo pi in apropinfo)
    {
        if(pi.CanRead && pi.GetGetMethod().IsStatic)
            iCount++;
    }

    // Allocate string arrays.
    astrLabels = new string[iCount];
    astrValues = new string[iCount];

    // Loop through the property information classes again.
    iCount = 0;
foreach (PropertyInfo pi in apropinfo)
{
    if (pi.CanRead && pi.GetGetMethod().IsStatic)
    {
        // Get the property names and values.
        astrLabels[iCount] = pi.Name;
        astrValues[iCount] = pi.GetValue(type, null).ToString();
        iCount++;
    }
}
Array.Sort(astrLabels, astrValues);
bValidInfo = true;

public static float MaxLabelWidth(Graphics grfx, Font font)
{
    return MaxWidth(Labels, grfx, font);
}
public static float MaxValueWidth(Graphics grfx, Font font)
{
    return MaxWidth(Values, grfx, font);
}
static float MaxWidth(string[] astr, Graphics grfx, Font font)
{
    float fMax = 0;

    GetSysInfo();

    foreach (string str in astr)
    {
        fMax = Math.Max(fMax, grfx.MeasureString(str, font).Width);
    }
    return fMax;
}

The `GetSysInfo` method in this class does the bulk of the work. It obtains the property names and their values when they are first required and whenever they change. The C# `typeof` operator obtains the type of the `SystemInformation` class, which is saved in a variable of type `Type`. One method of `Type` is `GetProperties`, which returns an array of `PropertyInfo` objects. Each object in this array is a property of `SystemInformation`. A `foreach` loop counts up all the properties that are both static and readable. (I know that all the properties of `SystemInformation` are static and readable today, but I'm trying to make the program generalized.)

The program then allocates arrays for the properties and their values, and loops through the `PropertyInfo` array again. The `Name` property of the `PropertyInfo` object is the name of the property; in this case, the `Name` property returns strings such as `HighContrast` and `IconSize`. The `GetValue` method obtains each property's value. The static `Sort` method of the `Array` class sorts both the name and value arrays based on the text of the property names.
The program that makes use of `SysInfoReflectionStrings` is functionally the same as the combination of SysInfoUpdate and SysInfoEfficient.

`SysInfoReflection.cs`

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoReflection: Form
{
    protected int iCount;
    protected string[] astrLabels;
    protected string[] astrValues;
    protected float cxCol;
    protected int cySpace;

    public static void Main()
    {
        Application.Run(new SysInfoReflection());
    }
    public SysInfoReflection()
    {
        Text = "System Information: Reflection";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        AutoScroll = true;

        SystemEvents.UserPreferenceChanged +=
            new UserPreferenceChangedEventArgs(UserPreferenceChanged);

        SystemEvents.DisplaySettingsChanged +=
            new EventHandler(DisplaySettingsChanged);

        UpdateAllInfo();
    }
    void UserPreferenceChanged(object obj, UserPreferenceChangedEventArgs ea)
    {
        UpdateAllInfo();
        Invalidate();
    }
    void DisplaySettingsChanged(object obj, EventArgs ea)
    {
```
UpdateAllInfo();
Invalidate();

void UpdateAllInfo()
{
    iCount     = SysInfoReflectionStrings.Count;
    astrLabels = SysInfoReflectionStrings.Labels;
    astrValues = SysInfoReflectionStrings.Values;

    Graphics grfx  = CreateGraphics();
    SizeF    sizef = grfx.MeasureString(" ", Font);
    cxCol   = sizef.Width +
              SysInfoReflectionStrings.MaxLabelWidth(grfx, Font);
    cySpace = Font.Height;

    AutoScrollMinSize = new Size(
        (int) Math.Ceiling(cxCol +
              SysInfoReflectionStrings.MaxValueWidth(grfx, Font)),
        (int) Math.Ceiling(cySpace * iCount));

    grfx.Dispose();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx  = pea.Graphics;
    Brush    brush = new SolidBrush(ForeColor);
    Point    pt    = AutoScrollPosition;

    int iFirst = (int)((pea.ClipRectangle.Top    - pt.Y) / cySpace);
    int iLast  = (int)((pea.ClipRectangle.Bottom - pt.Y) / cySpace);

    iLast = Math.Min(iCount - 1, iLast);

    for (int i = iFirst; i <= iLast; i++)
    {
        grfx.DrawString(astrLabels[i], Font, brush,
                         pt.X, pt.Y + i * cySpace);

        grfx.DrawString(astrValues[i], Font, brush,
                         pt.X + cxCol, pt.Y + i * cySpace);
    }
}
}
And this is my absolutely final version of a SystemInformation display program (until Chapter 6, that is, when I add a keyboard interface to it).
Chapter 5: Lines, Curves, and Area Fills

Overview

According to Euclid, "A line is breadthless length." It's the "breadthless" part that makes this statement interesting. It certainly indicates the high degree of abstract thought that characterizes ancient Greek mathematics. It also proves conclusively that the ancient Greeks knew nothing about computer graphics. They would have realized that pixels do indeed have breadth, a fact that contributes to one of the annoying problems often associated with computer graphics: the off-by-1-pixel error, a problem we'll be attuned to in this chapter.

The world of computer graphics is roughly divided into two areas:
- Vector graphics, which is the practical application of analytic geometry and involves drawing lines, curves, and filled areas
- Raster graphics, which involves bitmaps and real-world images

Then there's text, which occupies a plane of its own in the computer graphics world. In recent years, however, with the ascendancy of outline fonts, text is often treated as part of vector graphics.

This chapter introduces vector graphics as implemented in Microsoft Windows Forms and GDI+. All the drawing functions discussed in this chapter are methods of the Graphics class and begin with the prefix Draw or Fill. The Draw methods draw lines and curves; the Fill methods fill areas (the outlines of which, of course, are defined by lines and curves). The first argument to all the Draw methods covered in this chapter is a Pen object; the first argument to all the Fill methods is a Brush.


How to Get a Graphics Object

Most of the drawing functions are methods of the Graphics class. (There are additional drawing functions in the ControlPaint class, but these are somewhat specialized.) To draw, you need an object of type Graphics. But the Graphics constructor isn't public. You cannot simply create a Graphics object like so:
```
Graphics grfx = new Graphics(); // Won't work!
```

The Graphics class is also sealed, which means you can't derive your own class from Graphics:
```
class MyGraphics: Graphics // Won't work!
```

You must obtain the Graphics object in some other way. Here's a complete list of ways that you can do this, beginning with the most common:
- When you override the OnPaint method or install a Paint event handler in any class derived from Control (such as Form), a Graphics object is delivered to you as a property of the PaintEventArgs class.
- To paint on a control or a form at times other than during the OnPaint method or the Paint event, you can call the CreateGraphics method of the control. Classes sometimes call CreateGraphics in their constructors to obtain information and perform initialization. (Some of the programs in Chapter 4 did this.) Although classes can't draw during the constructor, they can do so during other events. It's common for a control or a form to draw something during keyboard, mouse, or timer events, as I'll demonstrate in Chapters 6, 8, and 10. It's important for a program to use the Graphics object only during the event that obtains it (that is, the Graphics object shouldn't be stored in a field in the class). The program should also call the Dispose method of the Graphics object when it's finished using it.
- When printing, you install a PrintPage event handler and get an object of type PrintPageEventArgs, which contains a Graphics object suitable for the printer. I'll demonstrate how to do this shortly.
- Some controls—most commonly menus, list boxes, combo boxes, and status bars—have a feature called owner draw that allows a program to dynamically draw items on the control. The MeasureItem and DrawItem events deliver objects of type MeasureItemEventArgs and DrawItemEventArgs, which have Graphics objects the event handler can use.
- To draw on a bitmap or a metafile (techniques I'll demonstrate in Chapters 11 and 23), you need to obtain a special Graphics object by calling the static method Graphics.FromImage.
If you need to obtain information from the Graphics object associated with a printer without actually printing, you can use the CreateMeasurementGraphics method of the PrinterSettings class.

If you're interfacing with Win32 code, you can use the static methods Graphics.FromHwnd and Graphics.FromHdc to obtain a Graphics object.

**Pens, Briefly**

When you draw a line by hand on paper, you use a pencil, a crayon, a fountain pen, a ballpoint pen, a felt-tip marker, or whatever. The type of instrument you choose at least determines the line's color and width. These qualities and others are subsumed under the Pen class, and whenever you draw a line, you specify a Pen object.

I’m not going to get into a comprehensive exploration of pens at this time, however. The problem is that pens can be created from brushes, so a complete discussion of pens is dependent on that topic. And brushes can be created from bitedmapped images and graphics paths, and those are more advanced graphics programming topics. We'll examine pens and brushes thoroughly in Chapter 17.

As I explained in Chapter 3, you can create a pen of a particular color like so:

```csharp
Pen pen = new Pen(color);
```

where `color` is an object of type `Color`. You can also take advantage of the Pens class, which contains 141 static read-only properties that return Pen objects. Pens.HotPink is thus an acceptable first argument to line-drawing methods (although appropriate only when used in moderation). A complete list of these color names is available on the inside back cover of this book.

The SystemPens class contains 15 static read-only properties that also return Pen objects based on the system colors. But if you want to create a pen that you know will be visible against the background color the user has chosen, base it on the current ForeColor property:

```csharp
Pen pen = new Pen(ForeColor);
```

There's one more aspect of pens I want to mention here, and that's the pen's width. The width is a read/write property:

<table>
<thead>
<tr>
<th>Pen Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>float</td>
</tr>
</tbody>
</table>

There's also a Pen constructor that includes the pen width, so here's a table listing the two Pen constructors I've mentioned so far:

<table>
<thead>
<tr>
<th>Pen Constructors (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pen(Color color)</td>
</tr>
<tr>
<td>Pen(Color color, float fWidth)</td>
</tr>
</tbody>
</table>

(Just so you won't think I'm holding back information, there are only two other Pen constructors, which look the same as these two except that a Brush object is the first argument.) When you use the first constructor, the pen is created with a width of 1. Pen objects available from the Pens and SystemPens class also have a width of 1. For the time being, you can think of that as 1 pixel wide. However, you'll discover in Chapter 7 that this width is actually in world coordinates and affected by various transforms.

It's possible to create a pen that is always 1 pixel wide regardless of any transforms that may be in effect. Use a width of 0 in the constructor:

```csharp
Pen pen = new Pen(color, 0);
```

Or set the Width property to 0:

```csharp
pen.Width = 0;
```
Straight Lines

To draw a single straight line, you use the DrawLine method of the Graphics class. There are four overloaded versions of DrawLine, but they all involve the same information: the coordinates where the line begins and ends, and the pen used to draw the line:

**Graphics DrawLine Methods**

void DrawLine(Pen pen, int x1, int y1, int x2, int y2)
void DrawLine(Pen pen, float x1, float y1, float x2, float y2)
void DrawLine(Pen pen, Point point1, Point point2)
void DrawLine(Pen pen, PointF point1, PointF point2)

You can specify the coordinates either as four int or float values, or as two Point or PointF structures.

DrawLine draws a line from the first point up to and including the second point. (This is a little different from Win32 GDI, which draws up to but not including the second point.) For example,
grfx.DrawLine(pen, 0, 0, 5, 5);

colors 6 pixels black—the pixels at coordinate points (0,0), (1,1), (2,2), (3,3), (4,4), and (5, 5). The order of the two points doesn't matter, so the call
grfx.DrawLine(pen, 5, 5, 0, 0);

produces identical results. The call
grfx.DrawLine(pen, 2, 2, 3, 3);

draws 2 pixels, at points (2, 2) and (3, 3). However,
grfx.DrawLine(pen, 3, 3, 3, 3);

draws nothing.

As you know, you can determine the width and height of your client area by using the ClientSize property of Form. The number of pixels horizontally is ClientSize.Width; the pixels can be numbered from 0 through ClientSize.Width – 1. Similarly, the vertical pixels can be numbered from 0 through ClientSize.Height – 1.

The XMarksTheSpot program draws an X in the client area.

**XMarksTheSpot.cs**

```csharp
// XMarksTheSpot.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class XMarksTheSpot: Form
{
    public static void Main()
    {
        Application.Run(new XMarksTheSpot());
    }
    public XMarksTheSpot()
```
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Pen      pen  = new Pen(ForeColor);

    grfx.DrawLine(pen, 0, 0,
                  ClientSize.Width - 1, ClientSize.Height - 1);
    grfx.DrawLine(pen, 0, ClientSize.Height - 1,
                  ClientSize.Width - 1, 0);
}

The first DrawLine call draws a line from the upper left pixel to the lower right pixel of the client area. The second DrawLine call begins the line at the lower left pixel, which is the point (0, ClientSize.Height−1), and goes to the upper right pixel at (ClientSize.Width−1, 0).

**An Introduction to Printing**

Many of the graphics demonstration programs in this chapter and subsequent chapters will be similar to XMarksTheSpot. They won't necessarily be as lame as XMarksTheSpot (although some will, unfortunately), but they'll do nothing more than demonstrate basic graphics programming techniques by displaying static images in their client areas.

It might be helpful even at this early stage to print these images on your printer as well, if only to have the option of proudly displaying them on your refrigerator door. More important, printing graphics gives you an opportunity to discover firsthand the extent to which a graphics programming system is device independent as you're learning the system.

Printing is a topic customarily banished to the end of programming books if not ignored entirely, mostly because printing is often terribly complicated. I've devoted Chapter 21 to printing to cover all the various facilities and options that are available. But for our immediate purposes—turning out a single page on the user's default printer using default settings—printing from a Windows Forms application is quite easy.

Indeed, the only reason I hesitated at all in introducing printing so early was the user-interface problem—how the program would allow the user to initiate printing. As you know, most programs that print include a Print option on the File menu. It's a little too early in the book for menus, a subject covered exhaustively in Chapter 14. I also considered implementing a simple keyboard interface, perhaps using the Print Screen (sometimes labeled PrtScn) key or Ctrl+P. Finally I decided on overriding the OnClick method.

OnClick is implemented in the Control class and inherited by every class descended from Control, including Form. The OnClick method is called whenever the user clicks the client area of the form with any mouse button. And that's all I'm going to say about the mouse until Chapter 8!

To print to the default printer, you first need to create an object of type PrintDocument, a class defined in the System.Drawing.Printing namespace:

PrintDocument prndoc = new PrintDocument();

We'll spend more time with this class in Chapter 21. For now, I'll mention only one property, one event, and one method in the class.
You set the `DocumentName` property of the `PrintDocument` object to a text string. This is the text that identifies the job in the printer dialog box as the graphics output is being spooled out to the printer:

```csharp
prndoc.DocumentName = "My print job";
```

A program that works with documents generally uses the name of the document for this text string. In this chapter, I'll use the program's caption bar text.

You need to create a method in your class that will perform the graphics output calls. This method is defined in accordance with the `PrintPageEventHandler` delegate. You can name this method `PrintDocumentOnPrintPage`, as in this example:

```csharp
void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
{
    :
}
```

Attach this handler to the `PrintPage` event of the `PrintDocument` object like so:

```csharp
prndoc.PrintPage += new PrintPageEventHandler(PrintDocumentOnPrintPage);
```

This is the same way that `Paint` event handlers were installed in some of the programs in Chapter 2 and in the SysInfoPanel program of Chapter 4.

To begin printing, the last thing you'll do with the `PrintDocument` object is call its `Print` method:

```csharp
prndoc.Print();
```

This `Print` method doesn't return immediately. Instead, a small message box is briefly displayed that contains the document name you specified and that gives the user the option of canceling the print job.

The `Print` method also causes your `PrintPage` event handler (which we've called `PrintDocumentOnPrintPage`) to be called. The `object` parameter to `PrintDocumentOnPrintPage` is the `PrintDocument` object created earlier. The `PrintPageEventArgs` parameter has properties that supply you with information about the printer. The most important of these properties, however, is named `Graphics` and is similar to the same-named property in `PaintEventArgs`, except that this property supplies you with a `Graphics` object for the printer page rather than for the client area of the form.

So the `PrintDocumentOnPrintPage` method often looks something like this:

```csharp
void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;
    :
}
```

You use that `Graphics` object to call methods that display graphics on the printer page.

If you were printing multiple pages, you'd set the `HasMorePages` property of `PrintPageEventArgs` to `true`, but because we're printing only one page, we leave the property at its default `false` setting and return from `PrintDocumentOnPrintPage`.

After `PrintDocumentOnPrintPage` returns with the `HasMorePages` property of the `PrintPageEventArgs` object set to `false`, the original call to the `Print` method of the `PrintDocument` object also returns. The program is finished with the print job. Sending the graphics output to the printer is somebody else's problem. Dealing with paper jams, empty ink cartridges, toner smudges, and bad cables is also somebody else's problem.

You might have more than one printer attached to your machine. The approach to printing that I've shown here uses your `default` printer. The Printers dialog box that you invoke from Control Panel or from the Settings submenu on your Start menu contains an item on its File menu to set the default printer.
As you know, a form's `ClientSize` property gives you the pixel dimensions of your form's client area, and that's sufficient for intelligently drawing within the client area. A similar property for the printer page is somewhat problematic.

A printer page is defined by three different areas. First, there's the entire size of the page. That information is provided by the `PageBounds` property of the `PrintPageEventArgs` class. It's a `Rectangle` structure where the X and Y properties are 0 and the Width and Height properties provide the default paper dimensions in units of 0.01 inch. For example, for 8½-by-11-inch paper, the Width and Height properties of `PageBounds` are equal to 850 and 1100. If the default printer settings indicate landscape rather than portrait, the Width and Height properties are set equal to 1100 and 850, respectively.

Second, the printable area of the page is usually very nearly the whole page except a margin where the print head (or whatever) can't reach. This margin may be different for the top and bottom of the page, and for the left and right. The `VisibleClipBounds` property of the `Graphics` class is a `RectangleF` structure that provides the size of the page's printable area. The X and Y properties of this structure are set to 0. The Width and Height properties indicate the dimensions of the printable area of the page in the same units that you'll be using for drawing to the printer.

The third area of the page takes into account 1-inch margins on all four edges of the page. Those represent bounds within which the user prefers to print. This information is returned in a `Rectangle` structure from the `MarginBounds` property of the `PrintPageEventArgs` object.

We'll explore these issues more in Chapter 21. For now, using the `VisibleClipBounds` property of the `Graphics` class is probably your best bet. The `Graphics` object you obtain from the `PrintPageEventArgs` object is consistent with this property—that is, the point (0, 0) references the upper left corner of the printable area of the page.

Of course, everything I so carefully emphasized about using a visible color on the video display is wrong for the printer. For a printer, the best color to use is `Color.Black`. The best pen is `Pens.Black`, and the best brush is `Brushes.Black`. That will take care of everybody except those strange people who insist on loading up their printers with black paper.

Here's a program that displays "Click to print" in its client area and prints when it gets a button click.

HelloPrinter.cs

```csharp
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class HelloPrinter : Form
{
    public static void Main()
    {
        Application.Run(new HelloPrinter());
    }
    public HelloPrinter()
    {
        Text = "Hello Printer!";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
}
```
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    StringFormat strfmt = new StringFormat();
    strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

    grfx.DrawString("Click to print", Font, new SolidBrush(ForeColor),
        ClientRectangle, strfmt);
}
protected override void OnClick(EventArgs ea)
{
    PrintDocument prndoc = new PrintDocument();
    prndoc.DocumentName = Text;
    prndoc.PrintPage += new PrintPageEventHandler(PrintDocumentOnPrintPage);
    prndoc.Print();
}
void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;

    grfx.DrawString(Text, Font, Brushes.Black, 0, 0);

    SizeF sizef = grfx.MeasureString(Text, Font);

    grfx.DrawLine(Pens.Black, sizef.ToPointF(),
        grfx.VisibleClipBounds.Size.ToPointF());
}

Notice that I've used the Text property of the form for both the print document name and as the text string argument to DrawString and MeasureString in the PrintDocumentOnPrintPage method. The program displays the text "Hello Printer!" in the upper left corner of the printable area of the page and then draws a line that extends from the bottom right corner of the text string to the bottom right corner of the printable area of the page. This example should be enough to assure you that VisibleClipBounds is indeed providing information consistent with the Graphics origin.

I'm detecting some scoffing among my readers. That's because I blithely used the form's Font property in the DrawString and MeasureString calls without considering that the printer has a resolution of 300, 600, 720, 1200, 1440, or perhaps even 2400 or 2880 dots per inch. The font accessible through the form's Font property was selected by the system to be appropriate for the video display, which probably has a resolution more like 100 dots per inch. The resultant text on the printer should therefore look quite puny.

Well go ahead: try it. The text is printed using a respectable 8-point font. Notice also that the diagonal line the program draws is obviously more than 1 pixel in width. One-pixel-wide lines on
today's high-resolution printers are barely visible. Windows Forms instead draws a nice solid line. Why that is so must remain a mystery for now, although a pleasant one. The answer will become apparent in Chapters 7 and 9.

Let's now write a program that displays the same output in both the form's client area and the printer page. I don't mean for you to copy and paste code from the OnPaint method to the PrintDocumentOnPrintPage method; let's demonstrate that we know something about programming by putting the graphics output code in a separate method named DoPage that is called by both OnPaint and PrintDocumentOnPrintPage. Here's a variation of XMarksTheSpot that does just that.

PrintForm.cs
//---------------------------------------------
// PrintForm.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintForm: Form
{
    public static void Main()
    {
        Application.Run(new PrintForm());
    }
    public PrintForm()
    {
        Text = "Printable Form";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        DoPage(pea.Graphics, ForeColor,
               ClientSize.Width, ClientSize.Height);
    }
    protected override void OnClick(EventArgs ea)
    {
        PrintDocument prndoc = new PrintDocument();

        prndoc.DocumentName = Text;
        prndoc.PrintPage +=
            new PrintPageEventHandler(PrintDocumentOnPrintPage);
        prndoc.Print();
    }
    void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics grfx = ppea.Graphics;
        DoPage(grfx, ForeColor,
               ClientSize.Width, ClientSize.Height);
    }
}
protected virtual void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Pen pen = new Pen(clr);

    grfx.DrawLine(pen, 0, 0, cx - 1, cy - 1);
    grfx.DrawLine(pen, cx - 1, 0, 0, cy - 1);
}

The `DoPage` method at the end of the listing outputs the graphics. The arguments are a `Graphics` object, a suitable color for the device, and the width and height of the output area. `DoPage` is called from two other methods: the `OnPaint` method and the `PrintDocumentOnPrintPage` method. In `OnPaint`, the last three `DoPage` arguments are set to `ForeColor` and the width and height of the form's client area. In `PrintDocumentOnPrintPage`, these arguments are `Color.Black` and the width and height of `VisibleClipBounds`.

I've given the last two arguments to `DoPage` names of `cx` and `cy`. The `c` stands for `count` and because `x` and `y` commonly refer to coordinates, `cx` and `cy` can be interpreted as referring to a "count" of the coordinate points, or the width and height.

Interestingly enough, when you have a `Graphics` object for your client area, the `VisibleClipBounds` property is equal to the width and height of the client area. I could have dispensed with the `cx` and `cy` arguments to `DoPage` and just used `VisibleClipBounds` inside `DoPage` for both the screen and the printer. However, I like having the width and height values already available in convenient variables, particularly considering what I've done here. Notice that I've made the `DoPage` method `protected` and `virtual`, and hence overridable. If you want to write a program that displays only a single screen of graphics, you can derive from `PrintableForm` rather than `Form` and have printing facilities built into your program.

And that's exactly what I'll do in virtually all the sample programs in this chapter and in many programs in subsequent graphics-oriented chapters.

Properties and State

Some graphics programming environments include the concept of a `current position`, which is a coordinate point retained by the environment that is used as a starting point in drawing functions. Generally, the graphics system defines one function to set the current position and another function to draw a line from the current position to a specified point. The drawing function also sets the current position to the new point.

GDI+ has no concept of a current position. This may come as a bit of a shock to veteran Windows programmers because drawing a line in Windows GDI requires two function calls, each of which specifies a single coordinate: `MoveTo` to set the current position to the specified point and `LineTo` to draw the line up to (but not including) that point.

GDI+ is also different from Windows GDI in that the `DrawLine` and `DrawString` calls include arguments specifying the font, the brush, and the pen. If GDI+ were more like Windows GDI, the font, the brush, and the pen would be properties of the `Graphics` object. You'll recall that `StringFormat` specifies certain details about the display of text. `StringFormat` is also an argument to `DrawString` rather than a property of the `Graphics` object.

For these reasons, the architects of GDI+ have termed it a `stateless` graphics programming environment. It's not `entirely` stateless, however. If it were, the `Graphics` class would have no read/write properties at all! As is, `Graphics` has 12 read/write properties as well as 6 read-only properties.
I count four settable properties of Graphics that have a profound impact on the appearance of graphics figures:

- PageScale and PageUnit determine the units you draw in. By default, you draw on the screen in units of pixels. I'll go over these properties in detail in Chapter 7.

- The Transform property is an object of type Matrix that defines a matrix transform for all graphics output. The transform translates, scales, shears, or rotates coordinate points. I'll cover the matrix transform in Chapter 7.

- Clip is a clipping region. When you set a clipping region, any drawing functions you call will be limited to output in that region. I'll discuss clipping in Chapter 15.

**Anti-Aliasing**

Besides those four properties of the Graphics class that affect output very profoundly, other properties affect the output in more subtle ways. Two of these properties are SmoothingMode and PixelOffsetMode.

<table>
<thead>
<tr>
<th>Graphics Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>SmoothingMode</td>
</tr>
<tr>
<td>PixelOffsetMode</td>
</tr>
</tbody>
</table>

These properties enable a graphics rendering technique known as anti-aliasing. The term alias in this context comes from sampling theory. Anti-aliasing is an attempt to reduce the sharp jaggies of displayed graphics by using shades of color.

Here's a program that draws a small line; I've also included statements to let you set the SmoothingMode and PixelOffsetMode properties.

**AntiAlias.cs**

```csharp
//----------------------------------------
// AntiAlias.cs © 2001 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class AntiAlias: Form
{
    public static void Main()
    {
        Application.Run(new AntiAlias());
    }

    public AntiAlias()
    {
        Text = "Anti-Alias Demo";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Pen pen = new Pen(ForeColor);
```
grfx.SmoothingMode = SmoothingMode.None;
grfx.PixelOffsetMode = PixelOffsetMode.Default;

ggfx.DrawLine(pen, 2, 2, 18, 10);
}
}

You can try various combinations of these properties, recompile and run the program, capture its screen image, and blow it up in some graphics or paint program to see the difference. Or you can sit back and let me do it.

By default, the line is rendered like so:

![Default Line](image)

I've included in this figure a little section of the form outside the client area on the left and top so you can clearly see that the line begins at pixel position (2, 2).

The `SmoothingMode` enumeration is defined in the namespace `System.Drawing.Drawing2D`:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>0</td>
<td>No anti-aliasing</td>
</tr>
<tr>
<td>HighSpeed</td>
<td>1</td>
<td>No anti-aliasing</td>
</tr>
<tr>
<td>HighQuality</td>
<td>2</td>
<td>Anti-aliasing enabled</td>
</tr>
<tr>
<td>None</td>
<td>3</td>
<td>No anti-aliasing</td>
</tr>
<tr>
<td>AntiAlias</td>
<td>4</td>
<td>Anti-aliasing enabled</td>
</tr>
<tr>
<td>Invalid</td>
<td>-1</td>
<td>Raises exception</td>
</tr>
</tbody>
</table>

There are really only three choices here: Do you want anti-aliasing? Do you not want it? Or would you rather raise an exception? The default is `None`.

When you enable anti-aliasing by setting the `SmoothingMode` property to `SmoothingMode.HighQuality` or `SmoothingMode.AntiAlias`, the line is rendered like so:

![Anti-Aliased Line](image)
It looks like a mess close up, but from a distance it's supposed to look smoother. (Not everyone
agrees: some people find that anti-aliasing makes things look "fuzzy.")

The idea here is that the line begins in the center of pixel (2, 2) and ends in the center of pixel (18,
10). The line is 1 pixel wide. When a graphics system uses anti-aliasing, the extent to which the
theoretical line intersects a pixel determines how black the pixel is colored.

If you enable anti-aliasing, you can enhance it a bit by using the `PixelOffsetMode` property. You set
the property to one of the `PixelOffsetMode` enumeration values, also (like `SmoothingMode`) defined
in `System.Drawing.Drawing2D`:

<table>
<thead>
<tr>
<th><code>PixelOffsetMode</code> Enumeration</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Default</code></td>
<td>0</td>
<td>Pixel offset not set</td>
</tr>
<tr>
<td><code>HighSpeed</code></td>
<td>1</td>
<td>Pixel offset not set</td>
</tr>
<tr>
<td><code>HighQuality</code></td>
<td>2</td>
<td>Half-pixel offset set</td>
</tr>
<tr>
<td><code>None</code></td>
<td>3</td>
<td>Pixel offset not set</td>
</tr>
<tr>
<td><code>Half</code></td>
<td>4</td>
<td>Half-pixel offset set</td>
</tr>
<tr>
<td><code>Invalid</code></td>
<td>-1</td>
<td>Raises exception</td>
</tr>
</tbody>
</table>

Again, you really have only three choices, and one of them is worthless. If you set the
`PixelOffsetMode` property to `Half` or `HighQuality`, the line is rendered like this:

Setting the pixel offset is more in accordance with an analytical geometry approach. The coordinates
of the line are decreased by half a pixel. The line is assumed to begin 2 pixels from the corner, which
is actually on the crack between the pixels.

**Multiple Connected Lines**

I mentioned earlier the concept of a current position used in some graphics programming
environments, and you may have thought such a thing odd because it implied that two function calls
were needed to draw a single line. Where the current position helps, however, is in drawing a series
of connected lines. Each additional call requires only one coordinate point. GDI+ isn't so economical.
Here, for example, are four `DrawLine` calls required to draw a box around the perimeter of the
program's client area:

```
grfx.DrawLine(pen, 0, 0, cx - 1, 0);
grfx.DrawLine(pen, cx - 1, 0, cx - 1, cy - 1);
grfx.DrawLine(pen, cx - 1, cy - 1, 0, cy - 1);
grfx.DrawLine(pen, 0, cy - 1, 0, 0);
```

Notice that the end point in each call must be repeated as the start point in the next call.

For this reason—and a couple other reasons that I'll discuss shortly—the `Graphics` class includes a
method to draw multiple connected lines, commonly called a polyline. The `DrawLines` (notice the
plural) method comes in two versions:

**Graphics DrawLines Methods**
void DrawLines(Pen pen, Point[] apt)
void DrawLines(Pen pen, PointF[] aptf)

You need an array of either integer Point coordinates or floating-point PointF coordinates.

Here’s the DrawLines code to outline the client area.

BoxingTheClient.cs
//----------------------------------------------
// BoxingTheClient.cs © 2001 by Charles Petzold
//----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BoxingTheClient: PrintableForm {
    public new static void Main() {
        Application.Run(new BoxingTheClient());
    }
    public BoxingTheClient() {
        Text = "Boxing the Client";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy) {
        Point[] apt = {new Point(0, 0),
                       new Point(cx - 1, 0),
                       new Point(cx - 1, cy - 1),
                       new Point(0, cy - 1),
                       new Point(0, 0)};

        grfx.DrawLines(new Pen(clr), apt);
    }
}

Notice that the class is derived from PrintableForm, so you can print it as well.

You can define the array of Point structures right in the DrawLines method. Here’s a program that does that. It’s the solution to a kids’ puzzle that involves drawing a particular design that resembles a house without lifting the pen or pencil from the paper.

DrawHouse.cs
//----------------------------------------
// DrawHouse.cs © 2001 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawHouse : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new DrawHouse());
    }
    public DrawHouse()
    {
        Text = "Draw a House in One Line";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawLines(new Pen(clr),
            new Point[]
            {
                new Point( cx / 4, 3 * cy / 4), // Lower left
                new Point( cx / 4, cy / 2),
                new Point( cx / 2, cy / 4), // Peak
                new Point(3 * cx / 4, cy / 2),
                new Point(3 * cx / 4, 3 * cy / 4), // Lower right
                new Point( cx / 4, cy / 2),
                new Point(3 * cx / 4, cy / 2),
                new Point( cx / 4, 3 * cy / 4), // Lower left
                new Point(3 * cx / 4, 3 * cy / 4) // Lower right
            });
    }
}

But the purpose of DrawLines isn't to solve kids' puzzles. In Chapter 17, you'll discover how you can create pens that are composed of patterns of dots and dashes, and how when you create thick pens, you can define the appearance of the ends of lines (whether they are rounded or square or whatnot) and the appearance of two lines that are joined together. These are called line ends and joins. In order for ends and joins to work correctly, GDI+ needs to know whether two lines that share a coordinate point are separate or connected. Using DrawLines rather than DrawLine is how you provide this information.

Another reason to use DrawLines is performance. This performance improvement is neither apparent nor important in the programs shown so far, but we haven't quite begun to exercise DrawLines. You see, the real purpose of DrawLines is not to draw straight lines. The real purpose is to draw curves. The trick is to make the individual lines very short and to use plenty of them. Any curve that you can define mathematically you can draw using DrawLines.

Don't hesitate to use hundreds or even thousands of Point or PointF structures in a single DrawLines call. That's what the function is for. Even a million Point or PointF structures passed to DrawLines doesn't take more than a second or two to render.
How many points do you need for a particular curve? Probably not a million. The curve will be smoothest if the number of points at least equals the number of pixels. You can often roughly approximate this number.

Here's some code that draws one cycle of a sine curve the size of the client area.

```csharp
// SineCurve.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class SineCurve: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new SineCurve());
    }
    public SineCurve()
    {
        Text = "Sine Curve";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        PointF[] aptf = new PointF[cx];
        for (int i = 0; i < cx; i++)
        {
            aptf[i].X = i;
            aptf[i].Y = cy / 2 * (1 - (float)Math.Sin(i * 2 * Math.PI / (cx - 1)));
        }
        grfx.DrawLines(new Pen(clr), aptf);
    }
}
```

This is the first program in this book to use a trigonometric method in the `Math` class, a very important class defined in the `System` namespace. I cover the `Math` class in more detail in Appendix B. The arguments to the trigonometric methods are in terms of radians rather than degrees. The `Math` class also includes two convenient `const` fields named `PI` and `E` that you can use with these methods. No longer will you need something like this at the top of your programs:

```
#define PI 3.14159     // Good riddance!
```

One note, however: most of the `Math` methods return `double` values; these must be explicitly cast to `float` before being used in `PointF` and similar structures.

It might be helpful to analyze in detail the assignment statement for the `Y` property of the `PointF` array: the argument to the `Math.Sin` function is in radians. One complete cycle (360°) is 2π radians. Thus, the argument ranges from 0 (when `i` is 0) to 2π (when `i` is `ClientSize.Width - 1`). The value of
the `Math.Sin` method ranges between −1 and +1. Normally, that value must be scaled by half the client area height to range from negative `ClientSize.Height / 2` to positive `ClientSize.Height / 2` and then subtracted from half the client area height to make the height range from 0 to `ClientSize.Height`. But I’ve used more efficient code by adding 1 to the negative result of the `Sin` method so that it ranges from 0 to 2 and then multiplying by half the client area height. Here’s what the result looks like:

![Sine Curve](image)

**Curves and Parametric Equations**

Coding a sine curve is relatively straightforward because values of $y$ are obtained by a simple function of $x$. In general, however, coding curves isn’t quite that simple. For example, the equation of the unit circle (that is, a circle with a radius of 1 unit) centered at the origin $(0, 0)$ is generally given as

$$x^2 + y^2 = 1$$

More generalized, a circle of radius $r$ can be expressed as

$$x^2 + y^2 = r^2$$

However, if you attempt to represent this equation in the form where $y$ is a function of $x$, you have

$$y = \pm \sqrt{r^2 - x^2}$$

There are several problems with this thing. The first is that $y$ has two values for every value of $x$. The second is that there are invalid values of $x$; $x$ must range between $-r$ and $+r$. A third, more practical, problem involves drawing a circle based on this equation. The resolution is lopsided: When $x$ is around 0, changes in $x$ produce relatively small changes in $y$. When $x$ approaches $r$ or $-r$, changes in $x$ produce much greater changes in $y$.

A more generalized approach to drawing curves uses parametric equations. In parametric equations, both the $x$ and $y$ coordinates of every point are calculated from functions based on a third variable, often called $t$. Intuitively, you can think of $t$ as time or some other abstract index necessary to define the entire curve. In graphics programming in Windows Forms, you can think of $t$ as ranging from 0 to one less than the number of `PointF` structures in your array.

The parametric equations that define a unit circle are

$$x(t) = \cos(t)$$
$$y(t) = \sin(t)$$
For $t$ ranging from 0 degrees to $2\pi$ degrees, these equations define a circle around the point $(0, 0)$ with a radius of 1.

The ellipse is defined similarly:

$$
\begin{align*}
  x(t) &= RX \cos(t) \\
  y(t) &= RY \sin(t)
\end{align*}
$$

The two axes of the ellipse are parallel to the horizontal and vertical axes. The horizontal ellipse axis is $2 \times RX$ in length; the vertical ellipse axis is $2 \times RY$. The ellipse is still centered around $(0, 0)$. To center it around the point $(CX, CY)$, the formulas are

$$
\begin{align*}
  x(t) &= CX + RX \cos(t) \\
  y(t) &= CY + RY \sin(t)
\end{align*}
$$

And here's a program to draw an ellipse that encompasses the full display area.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class PolyEllipse: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new PolyEllipse());
    }
    public PolyEllipse()
    {
        Text = "Ellipse with DrawLines";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        int iNum = 2 * (cx + cy);
        PointF[] aptf = new PointF[iNum];

        for (int i = 0; i < iNum; i++)
        {
            double dAng = i * 2 * Math.PI / (iNum - 1);
            aptf[i].X = (cx - 1) / 2f * (1 + (float)Math.Cos(dAng));
            aptf[i].Y = (cy - 1) / 2f * (1 + (float)Math.Sin(dAng));
        }
        grfx.DrawLines(new Pen(clr), aptf);
    }
}
```
Because the center of the ellipse is half the width and height of the display area, and the width and height of the ellipse are equal to the width and height of the display area, I was able to simplify the formulas a bit. I approximated the number of points in the array as the number of points that would be sufficient for a rectangle drawn around the display area.

You may have looked ahead in this chapter and discovered that the Graphics class includes a DrawEllipse method and wondered why we had to do one "manually." Well, that was just an exercise to prepare us for the next program, which draws something that certainly is not implemented by a simple method in Graphics.

Spiral.cs
//-------------------------------
// Spiral.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Spiral: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new Spiral());
    }
    public Spiral()
    {
        Text = "Spiral";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        const int iNumRevs   = 20;
        int       iNumPoints = iNumRevs * 2 * (cx + cy);
        PointF[]  aptf       = new PointF[iNumPoints];
float     fAngle, fScale;

for (int i = 0; i < iNumPoints; i++)
{
    fAngle = (float)(i * 2 * Math.PI / (iNumPoints / iNumRevs));
    fScale = 1 - (float)i / iNumPoints;

    aptf[i].X = (float)(cx / 2 * (1 + fScale * Math.Cos(fAngle)));
    aptf[i].Y = (float)(cy / 2 * (1 + fScale * Math.Sin(fAngle)));
}

gfx.DrawLines(new Pen(clr), aptf);

And here's what it looks like:

![Spiral Image]

**The Ubiquitous Rectangle**

Rectangles aren't found in nature very much, but they are certainly the most common form of objects designed and built by humans. Rectangles are everywhere. The page you're reading right now is a rectangle, these paragraphs are formatted into rectangles, the screenshot just before this section is a rectangle, the desk you're sitting at or the bed you're lying on is likely a rectangle, and the window you're gazing out of when I get a bit tedious is probably also a rectangle.

Certainly you can draw a rectangle using `DrawLine` or `DrawLines` (we've done it already when outlining the client area), but a simpler approach is the `DrawRectangle` method. In each of the three versions of `DrawRectangle`, a rectangle is defined by a point that specifies the upper left corner of the rectangle plus a width and a height. That's the same way the `Rectangle` structure is defined, and indeed, one of the methods uses that structure:

**Graphics DrawRectangle Methods**

```csharp
void DrawRectangle(Pen pen, int x, int y, int cx, int cy)
void DrawRectangle(Pen pen, float x, float y, float cx, float cy)
```
Oddly enough, there's no `DrawRectangle` method that uses a `RectangleF` structure. Perhaps a programmer forgot to mark it with a `public` modifier. Perhaps we'll see one in a later release.

The width and height of the rectangle must be greater than 0. Negative widths and heights won't raise exceptions, but nothing will be drawn.

When drawing rectangles, off-by-1 errors are common because the sides of the rectangles themselves are a pixel wide (at least). Does the width and height of the rectangle encompass the width of the sides, just one side, or neither side?

With default pen properties (an issue I'll talk about more in Chapter 17), a height and width of 3 in the dimensions passed to `DrawRectangle` results in this figure (blown up in size, of course):

![Rectangle](image)

The upper left corner of the figure is the pixel \((x, y)\). A width and height of 2 draws a 3-by-3-pixel rectangle with a single-pixel interior, as shown here:

![Rectangle](image)

A width and height of 1 causes a 2-by-2-pixel block to be drawn. You might be tempted to put the form's `ClientRectangle` property right in the `DrawRectangle` call

```csharp
grfx.DrawRectangle(pen, ClientRectangle);  // Avoid this!
```

to outline the outer visible edge of the client rectangle. It won't work! The right and bottom sides of the rectangle won't be visible. Next is a program that properly displays a complete rectangle on both the client area and the printer. I've made it red to be more visible on the screen.

**OutlineClientRectangle.cs**

```csharp
// OutlineClientRectangle.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class OutlineClientRectangle: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new OutlineClientRectangle());
    }
    
    public OutlineClientRectangle()
    {
        Text = "Client Rectangle";
    }
}
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.DrawRectangle(Pens.Red, 0, 0, cx - 1, cy - 1);
}

Try specifying only cx and cy, without subtracting 1, as the last two arguments of `DrawRectangle`. You'll note that the right and bottom edges of the rectangle won't be visible on the client area, which is the same problem as if you use the `ClientRectangle` property in the call to `DrawRectangle`.

The `Graphics` class also includes two methods for drawing multiple rectangles:

### Graphics DrawRectangles Methods

- void DrawRectangles (Pen pen, Rectangle[] arect)
- void DrawRectangles (Pen pen, RectangleF[] arectf)

These methods are much less useful than `DrawLines`. However, if you have a `RectangleF` structure named `rectf` (for example) and you want to draw a single rectangle based on that structure, and you then remember that no `DrawRectangle` overload is available for that structure, you can use `DrawRectangles` to do it:

```csharp
grfx.DrawRectangles(pen, new RectangleF[] { rectf });
```

### Generalized Polygons

Mathematically, polygons are closed figures of three or more sides, such as triangles, quadrilaterals, pentagons, hexagons, heptagons, octagons, nonagons, decagons, undecagons, dodecagons, and so forth. Here are two `Graphics` methods that draw polygons:

### Graphics DrawPolygon Methods

- void DrawPolygon(Pen pen, Point[] point)
- void DrawPolygon(Pen pen, PointF[] point)

The `DrawPolygon` method is very similar in functionality to `DrawLines`, except that the figure is automatically closed by a line that connects the last point to the first point. For example, consider the following array of `Point` structures:

```csharp
Point[] apt = {new Point (0, 0), new Point (50, 100), new Point (100, 0)};
```

The call

```csharp
grfx.DrawLines(pen, apt);
```

draws two lines that look like a V, and

```csharp
grfx.DrawPolygon(pen, apt);
```

draws a triangle.

In some cases, you could simulate a `DrawPolygon` call with a call to `DrawLines` and `DrawLine`:

```csharp
DrawLines(pen, apt);
DrawLine(pen, apt[apt.Length-1], apt[0]);
```

However, if you were dealing with wide lines with ends and joins, you wouldn't get exactly the same effect as with `DrawPolygon`. 
Easier Ellipses

We already know how to draw an ellipse using `DrawLines`, but here's an easier approach that takes the same arguments as `DrawRectangle`:

**Graphics DrawEllipse Methods**

```csharp
void DrawEllipse(Pen pen, int x, int y, int cx, int cy)
void DrawEllipse(Pen pen, float x, float y, float cx, float cy)
void DrawEllipse(Pen pen, Rectangle rect)
void DrawEllipse(Pen pen, RectangleF rectf)
```

The `DrawEllipse` methods are consistent with the `DrawRectangle` methods. For example, here's the ellipse drawn with a width and height of 3:

![Ellipse](image)

A width and height of 1 result in a solid 2-pixel-square figure.

What this means is that, as with `DrawRectangle`, to fit an ellipse in an area that is `cx` pixels wide and `cy` pixels high, you need to reduce the width and height by 1.

**ClientEllipse.cs**

```csharp
//-------------------------------
// ClientEllipse.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ClientEllipse: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new ClientEllipse());
    }
    public ClientEllipse()
    {
        Text = "Client Ellipse";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawEllipse(new Pen(clr), 0, 0, cx - 1, cy - 1);
    }
}
```
If the last two arguments of `DrawEllipse` are set to `cx` and `cy`, the right and bottom edges will be chopped off slightly.

**Arcs and Pies**

An arc—at least as far as Windows Forms is concerned—is a segment of an ellipse. To define an arc, you need to specify the same information as you need for an ellipse, plus you need to specify where the arc begins and where it ends. For that reason, each of the four versions of the `DrawArc` method require the same arguments as `DrawEllipse` plus two more arguments:

**Graphics DrawArc Methods**

```csharp
void DrawArc(Pen pen, int x, int y, int cx, int cy,
             int iAngleStart, int iAngleSweep)
void DrawArc(Pen pen, float x, float y, float cx, float cy,
             float fAngleStart, float fAngleSweep)
void DrawArc(Pen pen, Rectangle rect,
             float fAngleStart, float fAngleSweep)
void DrawArc(Pen pen, RectangleF rectf,
             float fAngleStart, float fAngleSweep)
```

These additional two arguments are angles that indicate the beginning of the arc and the length of the arc. The angles—which can be positive or negative—are measured clockwise in degrees beginning at the horizontal axis to the right of the ellipse’s center (that is, the position of 3:00 on a clock):

Here’s a program that draws an ellipse with a dashed circumference. The dashes are 10 degrees of arc; the gaps between the dashes are 5 degrees of arc.

**DashedEllipse.cs**

```csharp
// DashedEllipse.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class DashedEllipse : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new DashedEllipse());
    }
```
The dashed ellipse looks like this:

The Win32 API includes a function named RoundRect that draws a rectangle with rounded corners. The function takes four arguments that specify the upper left and lower right coordinates of the rectangle, plus two more arguments that specify the width and height of an ellipse that is used for curving the corners.

The Graphics class doesn't include a RoundRect method, but we can certainly attempt to simulate one.

RoundRect.cs
//-------------------------------
// RoundRect.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class RoundRect : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new RoundRect());
    }
    public RoundRect()
    {
        Text = "Rounded Rectangle";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        RoundedRectangle(grfx, Pens.Red,
                        new Rectangle(0, 0, cx - 1, cy - 1),
                        new Size(cx / 5, cy / 5));
    }
    void RoundedRectangle(Graphics grfx, Pen pen, Rectangle rect, Size size)
    {
        grfx.DrawLine(pen, rect.Left + size.Width / 2, rect.Top,
                      rect.Right - size.Width / 2, rect.Top);
        grfx.DrawArc(pen, rect.Right - size.Width, rect.Top,
                     size.Width, size.Height, 270, 90);
        grfx.DrawLine(pen, rect.Right, rect.Top + size.Height / 2,
                      rect.Right, rect.Bottom - size.Height / 2);
        grfx.DrawArc(pen, rect.Right - size.Width,
                     rect.Bottom - size.Height,
                     size.Width, size.Height, 0, 90);
        grfx.DrawLine(pen, rect.Right - size.Width / 2, rect.Bottom,
                      rect.Left + size.Width / 2, rect.Bottom);
        grfx.DrawArc(pen, rect.Right - size.Width,
                     rect.Bottom - size.Height,
                     size.Width, size.Height, 90, 90);
        grfx.DrawLine(pen, rect.Right - size.Width / 2, rect.Bottom - size.Height / 2,
                      rect.Left, rect.Top + size.Height / 2);
        grfx.DrawArc(pen, rect.Left, rect.Top,
                     size.Width, size.Height, 180, 90);
    }
}

The `RoundedRectangle` method I've written has a `Rectangle` argument that indicates the location and the size of the rectangle and a `Size` argument for the width and the height of an ellipse used to round the corners. I wrote the method to be consistent with the dimensions of the rectangle drawn by `DrawRectangle`—that is, when the width and height are set equal to 1 less than the width and height of the client area, the entire figure is visible. The method alternates `DrawLine` and `DrawArc` calls starting with the line at the top of the figure and continuing around clockwise.

I hesitate to recommend this as a general rounded rectangle drawing function, however. The individual lines and arcs are drawn with individual calls to `DrawLine` and `DrawArc`, which means that each of the eight pieces of the figure is drawn with line ends rather than line joins. The correct way to combine straight lines and curves into a single figure is with a graphics path. I'll show you how in Chapter 15.

The `DrawPie` methods have the same arguments as `DrawArc`, but these methods also draw lines from the ends of the arc to the center of the ellipse, creating an enclosed area:

**Graphics DrawPie Methods**

```csharp
void DrawPie(Pen pen, int x, int y, int cx, int cy,
             int iAngleStart, int iAngleSweep)
void DrawPie(Pen pen, float x, float y, float cx, float cy,
             float fAngleStart, float fAngleSweep)
void DrawPie(Pen pen, Rectangle rect,
             float fAngleStart, float fAngleSweep)
void DrawPie(Pen pen, RectangleF rectf,
             float fAngleStart, float fAngleSweep)
```

The pie chart is, of course, a venerable fixture in business graphics. The problem is, if you really need to code up a pie chart, you probably want to adorn it with 3-D effects and such, which means that `DrawPie` provides less convenience than you might think. Regardless, here's a program that draws a pie chart based on an array of values (stored as a field) that I made up for this purpose.

**PieChart.cs**

```csharp
// PieChart.cs © 2001 by Charles Petzold
//-------------------------------
```
using System;
using System.Drawing;
using System.Windows.Forms;

class PieChart: PrintableForm
{
    int[] aiValues = { 50, 100, 25, 150, 100, 75 };;

    public new static void Main()
    {
        Application.Run(new PieChart());
    }
    public PieChart()
    {
        Text = "Pie Chart";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Rectangle rect = new Rectangle(50, 50, 200, 200);
        Pen pen = new Pen(clr);
        int iTotal = 0;
        float fAngle = 0, fSweep;

        foreach (int iValue in aiValues)
        {
            iTotal += iValue;
        }

        foreach (int iValue in aiValues)
        {
            fSweep = 360f * iValue / iTotal;
            DrawPieSlice(grfx, pen, rect, fAngle, fSweep);
            fAngle += fSweep;
        }
    }

    protected virtual void DrawPieSlice(Graphics grfx, Pen pen, Rectangle rect, float fAngle, float fSweep)
    {
        grfx.DrawPie(pen, rect, fAngle, fSweep);
    }
}

Notice the Rectangle definition in the DoPage method. This is the only program in this chapter that uses absolute coordinates and sizes, the reason being that elliptical pie charts aren't very attractive. The DoPage method totals the array of values and then calculates a sweep angle for each slice by dividing the value by the total and multiplying by 360 degrees. Here's the result:
I'm sorry, but I just can't let you think that this is the best pie chart I can come up with! Fortunately, I was prescient enough to isolate the call to `DrawPie` in a virtual function in `PieChart`. That makes it easy to override this method in a BetterPieChart program.

**BetterPieChart.cs**

```csharp
// BetterPieChart.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterPieChart : PieChart
{
    public new static void Main()
    {
        Application.Run(new BetterPieChart());
    }
    public BetterPieChart()
    {
        Text = "Better " + Text;
    }
    protected override void DrawPieSlice(Graphics grfx, Pen pen, Rectangle rect, float fAngle, float fSweep)
    {
        float fSlice = (float)(2 * Math.PI * (fAngle + fSweep / 2) / 360);
    }
}```
The `fSlice` variable is the angle of the center of the slice converted to radians. I use that to calculate $x$ and $y$ offset values that are applied to the rectangle that defines the size and location of the pie slices. The result is that each slice is moved away from the center for an "exploded" view:

This doesn't exhaust the collection of line-drawing methods in the `Graphics` class. You can draw curves more complex than elliptical arcs by using the `DrawBezier`, `DrawBeziers`, `DrawCurve`, and `DrawClosedCurve` methods that you'll find out about in Chapter 11. You can assemble a collection of lines and curves into a graphics path and render that path using the `DrawPath` method. We'll get to that topic in Chapter 15.

**Filling Rectangles, Ellipses, and Pies**

Several of the `Graphics` methods discussed so far have defined enclosed areas, even though these methods have only drawn the outline of the area with the specified pen and not filled the interior of the area. For those methods prefixed with `Draw` that define enclosed areas, there are also methods beginning with `Fill` that fill the interiors. The first argument to these methods is the `Brush` used to fill the area.

Here are the four versions of the `FillRectangle` method:

**Graphics FillRectangle Methods**

```csharp
void FillRectangle(Brush brush, int x, int y, int cx, int cy)
void FillRectangle(Brush brush, float x, float y, float cx, float cy)
void FillRectangle(Brush brush, Rectangle rect)
void FillRectangle(Brush brush, RectangleF rectf)
```

The width and height of the resultant figure is equal to the width and height specified in the method arguments. For example, if the width and height are equal to 3, the `FillRectangle` call draws a 3-pixel-
square block with the upper left corner at pixel \((x, y)\). If you want to draw and fill a particular rectangle, call \textit{FillRectangle} first so the fill doesn't overwrite any of the lines.

The \textit{Graphics} class also includes two \textit{FillRectangles} methods:

\textbf{Graphics FillRectangles Methods}

\begin{verbatim}
void FillRectangles(Brush brush, Rectangle[] arect)
void FillRectangles(Brush brush, RectangleF[] arect)
\end{verbatim}

These \textit{FillRectangles} methods produce the same results as multiple calls to \textit{FillRectangle}.

There are four \textit{FillEllipse} methods, and they have the same arguments as \textit{DrawEllipse}:

\textbf{Graphics FillEllipse Methods}

\begin{verbatim}
void FillEllipse(Brush brush, int x, int y, int cx, int cy)
void FillEllipse(Brush brush, float x, float y, float cx, float cy)
void FillEllipse(Brush brush, Rectangle rect)
void FillEllipse(Brush brush, RectangleF rectf)
\end{verbatim}

\textit{FillEllipse} behaves a little differently from all the methods covered so far. For example, suppose you specify a location of \((0, 0)\) and a height and width of 20 for the ellipse. As you know, \textit{DrawEllipse} draws a figure that encompasses pixels 0 through 20 both horizontally and vertically for an effective width and height of 21 pixels.

For the most part, the area colored by \textit{FillEllipse} encompasses pixels 1 through 19 both horizontally and vertically, for an effective width of 19 pixels. I say "for the most part" because there always seems to be 1 pixel at the left that occupies pixel position 0! There's also some overlap between the ellipse drawn by \textit{DrawEllipse} and the area filled by \textit{FillEllipse}. If you need to draw an ellipse that is both filled and outlined, call \textit{FillEllipse} before calling \textit{DrawEllipse}.

There are also three \textit{FillPie} methods:

\textbf{Graphics FillPie Methods}

\begin{verbatim}
void FillPie(Brush brush, int x, int y, int cx, int cy,
            int iAngle, int iSweep)
void FillPie(Brush brush, float x, float y, float cx, float cy,
            float fAngle, float fSweep)
void FillPie(Brush brush, Rectangle rect,
            float fAngle, float fSweep)
\end{verbatim}

\textbf{Off by 1}

Now that we've examined all the rectangle and ellipse methods, it's time to compare them with the purpose of avoiding off-by-1 errors. The following program draws \(4 \times 4\) rectangles and ellipses using the six methods \textit{DrawRectangle}, \textit{DrawRectangles}, \textit{DrawEllipse}, \textit{FillRectangle}, \textit{FillRectangles}, and \textit{FillEllipse}.

\texttt{FourByFours.cs}

\begin{verbatim}
// FourByFours.cs © 2001 by Charles Petzold
//------------------------------
\end{verbatim}
using System;
using System.Drawing;
using System.Windows.Forms;

class FourByFours: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FourByFours());
    }
    public FourByFours()
    {
        Text = "Four by Fours";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr);
        Brush brush = new SolidBrush(clr);

        grfx.DrawRectangle(pen, new Rectangle(2, 2, 4, 4));
        grfx.DrawRectangles(pen, new Rectangle[]
                {new Rectangle(8, 2, 4, 4)});
        grfx.DrawEllipse(pen, new Rectangle(14, 2, 4, 4));
        grfx.FillRectangle(brush, new Rectangle(2, 8, 4, 4));
        grfx.FillRectangles(brush, new Rectangle[]
                {new Rectangle(8, 8, 4, 4)});
        grfx.FillEllipse(brush, new Rectangle(14, 8, 4, 4));
    }
}

Here's what the output looks like blown up to analyzable size:

As you can see, the DrawRectangle, DrawRectangles, and DrawEllipse methods are all consistent in rendering figures that are an extra pixel wider and higher than the size would imply. With the
exception of a little nub on the left, the FillEllipse method draws a figure that is a pixel narrower and shorter than the 4 × 4 figures drawn by FillRectangle and FillRectangles.

**Polygons and the Filling Mode**

Finally (at least for this chapter), we have the FillPolygon method. What makes the polygon different from other filled areas is that the lines that define the polygon can cross and overlap. This adds a layer of complexity because the interiors of the polygon can be filled in two distinct ways. There are four FillPolygon methods:

**Graphics FillPolygon Methods**

void FillPolygon(Brush brush, Point[] apt)
void FillPolygon(Brush brush, PointF[] apt)
void FillPolygon(Brush brush, Point[] apt, FillMode fm)
void FillPolygon(Brush brush, PointF[] apt, FillMode fm)

These are similar to the DrawPolygon methods except that an optional argument is included. FillMode is an enumeration defined in the namespace System.Drawing.Drawing2D with just two possible values:

**FillMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternate</td>
<td>0</td>
<td>Default; alternates filled and unfilled areas</td>
</tr>
<tr>
<td>Winding</td>
<td>1</td>
<td>Most interior areas filled</td>
</tr>
</tbody>
</table>

The fill mode makes a difference only when the lines that define the polygon overlap. The fill mode determines which of the enclosed areas are filled and which are not. If you don't specify a fill mode in the FillPolygon method, FillMode.Alternate is the default. In this case, an enclosed area is filled only if there are an odd number of boundaries between the enclosed area and infinity.

The classical example is the five-point star. The interior pentagon is filled when the winding fill mode is used but not when the alternate fill mode is used.

**FillModesClassical.cs**

```csharp
//-------------------------------------------------
// FillModesClassical.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class FillModesClassical: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FillModesClassical());
    }
    public FillModesClassical()
    {
```
Alternate and Winding Fill Modes (The Classical Example);

ClientSize = new Size(2 * ClientSize.Height, ClientSize.Height);

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Brush brush = new SolidBrush(clr);
    Point[] apt = new Point[5];

    for (int i = 0; i < apt.Length; i++)
    {
        double dAngle = (i * 0.8 - 0.5) * Math.PI;
        apt[i] = new Point(
            (int)(cx *(0.25 + 0.24 * Math.Cos(dAngle))),
            (int)(cy *(0.50 + 0.48 * Math.Sin(dAngle))));
    }

    grfx.FillPolygon(brush, apt, FillMode.Alternate);

    for (int i = 0; i < apt.Length; i++)
        apt[i].X += cx / 2;

    grfx.FillPolygon(brush, apt, FillMode.Winding);
}

The first for loop defines the five points of the star displayed in the left half of the client area. That polygon is filled with the alternate fill mode. The second for loop shifts the points to the right side of the client area where the polygon is filled with the winding fill mode.

In most cases, the winding fill mode causes all enclosed areas to be filled. But it's not quite that simple, and there are exceptions. To determine whether an enclosed area is filled in winding mode, imagine a line drawn from a point in that area to infinity. If the imaginary line crosses an odd number of boundary lines, the area is filled, just as in alternate mode. If the imaginary line crosses an even number of boundary lines, the area can be either filled or not filled. The area is filled if the number of boundary lines going in one direction (relative to the imaginary line) is not equal to the number of boundary lines going in the other direction.

With a little effort, it's possible to devise a figure that leaves an interior unfilled with winding mode. FillModesOddity.cs
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class FillModesOddity : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FillModesOddity());
    }
    public FillModesOddity()
    {
        Text = "Alternate and Winding Fill Modes (An Oddity)";
        ClientSize = new Size(2 * ClientSize.Height, ClientSize.Height);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        PointF[] aptf = { new PointF(0.1f, 0.7f), new PointF(0.5f, 0.7f),
                         new PointF(0.5f, 0.1f), new PointF(0.9f, 0.1f),
                         new PointF(0.9f, 0.5f), new PointF(0.3f, 0.5f),
                         new PointF(0.3f, 0.9f), new PointF(0.7f, 0.9f),
                         new PointF(0.7f, 0.3f), new PointF(0.1f, 0.3f)};
        for (int i = 0; i < aptf.Length; i++)
        {
            aptf[i].X *= cx / 2;
            aptf[i].Y *= cy;
        }
        grfx.FillPolygon(brush, aptf, FillMode.Alternate);
        for (int i = 0; i < aptf.Length; i++)
            aptf[i].X += cx / 2;
        grfx.FillPolygon(brush, aptf, FillMode.Winding);
    }
}
I'll discuss three more Fill methods in subsequent chapters: FillClosedCurve in Chapter 13, and FillRegion and FillPath in Chapter 15.
Chapter 6: Tapping into the Keyboard

Overview

Despite the sophisticated, visually oriented point-and-click user interface of today's graphical environments—including the mouse, controls, menus, and dialog boxes—the keyboard remains the primary source of input in most applications. The keyboard also has the most ancient ancestry of any component of the personal computer, dating from 1874 with the first Remington typewriter. Through a few decades of evolution, computer keyboards have expanded beyond the typewriter to include function keys, cursor positioning keys, and (usually) a separate numeric keypad.

In most computers, the keyboard and the mouse are the sole sources of user input. While much research continues with voice and handwriting recognition for entering alphanumeric data into the computer, those input methods don't—and never will—offer the precision of the keyboard. (Of course, I'm assuming that computers will never be better than humans at decoding bad handwriting or strong accents.)

Ignoring the Keyboard

Although the keyboard is an important source of user input in most applications, you certainly don't need to write code that acts on each and every keyboard event your application receives. Microsoft Windows and the Windows Forms class libraries handle many keyboard functions themselves.

Applications can usually ignore keystrokes involved in menu selection, for example. Programs don't need to monitor those keystrokes because the menu itself handles all the keyboard activity associated with menu selection and tells your program—through an event—when a menu item has been selected.

Windows Forms programs often define keyboard accelerators to invoke common menu items. These accelerators usually involve the Ctrl key in combination with a letter or function key (for example, Ctrl+S to save a file). Again, your application doesn't have to worry about translating these keystrokes into menu commands. The menu itself does that.

Dialog boxes also have a keyboard interface, but programs usually don't need to monitor the keyboard when a dialog box is active. Your program is notified through events of the effects of any keystrokes in the dialog box. If you put controls on the surface of your form, you don't need to worry about navigation keys, such as Tab or the cursor-movement keys (other than to ensure at design time that the tab order is logical); all user navigation through the controls is handled for you. You can also use controls such as TextBox and RichTextBox to process keyboard input. These controls deliver a resultant text string to your program when the user has finished entering the input.

Despite all this help, there remain many applications in which you need to process keyboard input directly. Certainly if you're going to be writing your own controls, you need to know something about the keyboard.

Who's Got the Focus?

The keyboard is a shared resource in Windows. All applications get input from the same keyboard, yet any particular keystroke must have only a single destination. For Windows Forms programs, this destination must be an object of type Control (the class that implements keyboard handling) or a descendant of Control, such as Form.

The object that receives a particular keyboard event is the object that has the input focus. The concept of input focus is closely related to the concept of the active form. The active form is usually easy to identify. It is often the topmost form on the desktop. If the active form has a caption bar, the caption bar is highlighted. If the active form has a dialog box frame instead, the frame is highlighted. If the active form is currently minimized, its entry in the taskbar is shown as a depressed button.

The active form is available from the only static property implemented by Form:

<table>
<thead>
<tr>
<th>Form Static Property</th>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Form</td>
<td>ActiveForm</td>
<td>get</td>
</tr>
</tbody>
</table>
However, this property returns a non-null object only if the currently active form is part of your application. It can't obtain objects created by other applications!

A form can attempt to make itself the active form by calling the following method:

**Form Methods (selection)**

```csharp
void Activate()
```

Usually, if the form is not topmost on the desktop, this call will cause Windows to flash the form's entry in the taskbar, requiring the user to bring the form topmost and make it the active form. In addition, the Form class implements two events related to the active form:

**Form Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activated</td>
<td>OnActivated</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Deactivate</td>
<td>OnDeactivate</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

I'm mentioning these properties, methods, and events now because you probably won't be using them much. It isn't often necessary for a program to get involved with the activation of its forms.

Input focus is another matter. If the active form has child controls—that is, controls on the surface of its client area, like the Panel control in Chapter 4—the object with the input focus must be either one of these controls or the form itself. Controls indicate they have input focus in different ways. A button displays a dotted line around the text; a text box displays a flashing caret. I'll discuss issues related to input focus later in this chapter; they will surface again in subsequent chapters as well.

### Keys and Characters

A keyboard always generates numeric codes of some sort. But you can think of a keyboard in two different ways:

- As a collection of distinct physical keys
- As a means of generating character codes

When you treat the keyboard as a collection of keys, any code generated by the keyboard must identify the key and indicate whether the key is being pressed or released. When you treat the keyboard as a character input device, the code generated by a particular keystroke identifies a unique character in a character set. Traditionally, this character set has been ASCII, but in the Windows Forms environment the character set is Unicode.

Many of the keys on today's computer keyboards aren't associated with character codes. Neither the function keys nor the cursor-movement keys generate character codes. Programs that use keyboard input in any nontrivial manner usually must deal with the keyboard as both a collection of keys and a character generator.

You can think of the keyboard as divided into four general groups of keys:

- **Toggle keys** Caps Lock, Num Lock, Scroll Lock, and possibly the Insert key. Pressing the key is intended to turn on the state of the key; pressing it again turns the state off. The Caps Lock, Num Lock, and Scroll Lock keys have systemwide states. (That is, when programs are running concurrently on the same computer, Caps Lock can't be simultaneously on for one program and off for another.) Keyboards often have lights that indicate the toggle state of these keys.

- **Shift keys** The Shift, Ctrl, and Alt keys. When depressed, the shift keys affect the interpretation of other keys. The shift keys are called modifier keys in the Windows Forms class library.

- **Noncharacter keys** The function keys, the cursor movement keys, Pause, Delete, and possibly the Insert key. These keys aren't associated with characters but instead often direct a program to carry out a particular action.

- **Character keys** The letter, number, and symbol keys, the spacebar, the Tab key, Backspace, Esc, and Enter. (The Tab, Backspace, Esc, and Enter keys can also be treated as noncharacter keys.)
Often a single physical key can generate different character codes depending on the state of the toggle and shift keys. For example, the A key generates a lowercase a or an uppercase A depending on the Caps Lock and Shift keys. Sometimes two different physical keys (such as the two Enter keys on most personal computer keyboards) can generate the same character code.

A Windows Forms program receives keyboard input in the form of events. I'll describe first how to treat the keyboard as a collection of keys and then how to treat it as a generator of character codes.

**Keys Down and Keys Up**

Much of the keyboard functionality is implemented in the Control class, which supports the following two events and methods that let you deal with key-down and key-up events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyDown</td>
<td>OnKeyDown</td>
<td>KeyEventHandler</td>
<td>KeyEventArgs</td>
</tr>
<tr>
<td>KeyUp</td>
<td>OnKeyUp</td>
<td>KeyEventHandler</td>
<td>KeyEventArgs</td>
</tr>
</tbody>
</table>

As usual, in any class derived from Control (such as Form), you can override the OnKeyDown and OnKeyUp methods:

```csharp
protected override void OnKeyDown(KeyEventArgs kea)
{
    :
}
protected override void OnKeyUp(KeyEventArgs kea)
{
    :
}
```

This is the customary way of handling key events in a class derived from Form.

You can also process key-down and key-up events in objects created from the Control class or one of its descendents. You first need to define methods in accordance with the KeyEventHandler delegate:

```csharp
void MyKeyDownHandler(object objSender, KeyEventArgs kea)
{
    :
}
void MyKeyUpHandler(object objSender, KeyEventArgs kea)
{
    :
}
```

You then register the key event handlers:

```csharp
cntl.KeyDown += new KeyEventHandler(MyKeyDownHandler);
cntl.KeyUp   += new KeyEventHandler(MyKeyUpHandler);
```

Whichever way you do it, you get a KeyEventArgs object when a key is pressed or released. This object has the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keys</td>
<td>KeyCode</td>
<td>get</td>
<td>Identifies the key</td>
</tr>
</tbody>
</table>
There's a whole lot of redundancy here. The only necessary properties are `KeyData` and `Handled`. Everything else can be derived from `KeyData`. But the redundancy is convenient. You'll probably find yourself using the `KeyCode`, `Shift`, `Control`, and `Alt` properties the most.

The first three properties in this table are all of the same type—a very important enumeration named `Keys`. The `KeyCode` property tells you what key is being pressed; that's the most important information. The `Modifiers` property indicates whether the Alt, Ctrl, or Shift keys are also pressed. `KeyData` combines `KeyCode` and `Modifiers`; `Shift`, `Control`, and `Alt` duplicate the `Modifiers` information in Booleans. `Handled` is a property sometimes set to `true` by controls to indicate that the control has used a keyboard event and it shouldn't be passed to the control's parent. `KeyValue` returns the same information as `KeyData` but as an integer rather than as a `Keys` enumeration.

**The Keys Enumeration**

Three of the properties of `KeyEventArgs` are of type `Keys`. `Keys` is a large enumeration—the second largest enumeration in the entire .NET Framework. It includes keys that certainly aren't on my keyboard and probably aren't on yours either. (Veteran Windows programmers might notice that these enumeration values are the same as the virtual key codes defined in the Windows header files.)

Let's tackle the `Keys` enumeration in logical groups. First, `Keys` has 26 members that identify the letter keys regardless of the shift state:
Notice that the enumeration values are the same as the ASCII codes (which are the same as the Unicode codes) for the uppercase letters. (These keys also generate character codes that are dependent on the Ctrl, Shift, and Caps Lock states.)

Just so we don’t get too far adrift here, let’s look at some code that makes use of one of the `Keys` values.

**ExitOnX.cs**

```csharp
// ExitOnX.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class ExitOnX: Form
{
    public static void Main()
    {
        Application.Run(new ExitOnX());
    }

    public ExitOnX()
    {
        Text = "Exit on X";
        BackColor = SystemColors.Window;
       ForeColor = SystemColors.WindowText;
    }

    protected override void OnKeyDown(KeyEventArgs kea)
    {
        if (kea.KeyCode == Keys.X)
        
        Close();
    }
}
```

This program closes itself when you press the X key. You can have any combination of Shift, Ctrl, or Alt keys pressed when you press the X. Because you know the relationship between the enumeration values and Unicode, the `if` statement could be replaced by

```csharp
if (kea.KeyCode == (Keys)(int)'X')
```

The next set of `Keys` values refers to the horizontal row of number keys located above the letter keys regardless of shift state:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>48</td>
</tr>
<tr>
<td>D1</td>
<td>49</td>
</tr>
<tr>
<td>D2</td>
<td>50</td>
</tr>
<tr>
<td>D3</td>
<td>51</td>
</tr>
<tr>
<td>D4</td>
<td>52</td>
</tr>
</tbody>
</table>
Keys Enumeration (number keys)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D5</td>
<td>53</td>
</tr>
<tr>
<td>D6</td>
<td>54</td>
</tr>
<tr>
<td>D7</td>
<td>55</td>
</tr>
<tr>
<td>D8</td>
<td>56</td>
</tr>
<tr>
<td>D9</td>
<td>57</td>
</tr>
</tbody>
</table>

Again, notice the relationship to the ASCII codes for the numbers. These keys also generate character codes that depend on the shift state.

The Keys enumeration has values for 24 function keys:

Keys Enumeration (function keys)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>112</td>
<td>F13</td>
<td>124</td>
</tr>
<tr>
<td>F2</td>
<td>113</td>
<td>F14</td>
<td>125</td>
</tr>
<tr>
<td>F3</td>
<td>114</td>
<td>F15</td>
<td>126</td>
</tr>
<tr>
<td>F4</td>
<td>115</td>
<td>F16</td>
<td>127</td>
</tr>
<tr>
<td>F5</td>
<td>116</td>
<td>F17</td>
<td>128</td>
</tr>
<tr>
<td>F6</td>
<td>117</td>
<td>F18</td>
<td>129</td>
</tr>
<tr>
<td>F7</td>
<td>118</td>
<td>F19</td>
<td>130</td>
</tr>
<tr>
<td>F8</td>
<td>119</td>
<td>F20</td>
<td>131</td>
</tr>
<tr>
<td>F9</td>
<td>120</td>
<td>F21</td>
<td>132</td>
</tr>
<tr>
<td>F10</td>
<td>121</td>
<td>F22</td>
<td>133</td>
</tr>
<tr>
<td>F11</td>
<td>122</td>
<td>F23</td>
<td>134</td>
</tr>
<tr>
<td>F12</td>
<td>123</td>
<td>F24</td>
<td>135</td>
</tr>
</tbody>
</table>

I know: I don't have 24 function keys on my keyboard either, and I think I prefer it that way.

The original IBM PC keyboard introduced a numeric keypad. The following keys of the numeric keypad generate the same codes regardless of the Num Lock state:

Keys Enumeration (keypad operators)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>106</td>
<td>Numeric keypad *</td>
</tr>
<tr>
<td>Add</td>
<td>107</td>
<td>Numeric keypad +</td>
</tr>
<tr>
<td>Subtract</td>
<td>109</td>
<td>Numeric keypad –</td>
</tr>
<tr>
<td>Divide</td>
<td>111</td>
<td>Numeric keypad /</td>
</tr>
</tbody>
</table>

The other keys of the numeric keypad generate different codes depending on the state of the Num Lock key. Here are the numeric keypad codes when Num Lock is toggled on. I've arranged this table somewhat like the numeric keypad itself:

Keys Enumeration (keypad numbers)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
</table>
### Keys Enumeration (keypad numbers)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPad7</td>
<td>103</td>
<td>NumPad8</td>
<td>104</td>
<td>NumPad9</td>
<td>105</td>
</tr>
<tr>
<td>NumPad4</td>
<td>100</td>
<td>NumPad5</td>
<td>101</td>
<td>NumPad6</td>
<td>106</td>
</tr>
<tr>
<td>NumPad1</td>
<td>97</td>
<td>NumPad2</td>
<td>98</td>
<td>NumPad3</td>
<td>99</td>
</tr>
<tr>
<td>NumPad0</td>
<td>96</td>
<td>Decimal</td>
<td>110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These keys also generate character codes for the 10 digits and the decimal separator character. For keyboard layouts in some countries, the decimal separator character is a period. In others, it's a comma. Regardless, the KeyCode value is `Keys.Decimal`. The following enumeration value doesn't seem to be used:

### Keys Enumeration (keypad, unused)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separator</td>
<td>108</td>
</tr>
</tbody>
</table>

Here are the codes generated when Num Lock is toggled off:

### Keys Enumeration (keypad cursor movement)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>36</td>
<td>Up</td>
<td>38</td>
<td>PageUp or Prior</td>
<td>33</td>
</tr>
<tr>
<td>Left</td>
<td>37</td>
<td>Clear</td>
<td>12</td>
<td>Right</td>
<td>39</td>
</tr>
<tr>
<td>End</td>
<td>35</td>
<td>Down</td>
<td>40</td>
<td>PageDown or Next</td>
<td>34</td>
</tr>
<tr>
<td>Insert</td>
<td>45</td>
<td></td>
<td></td>
<td>Delete</td>
<td>46</td>
</tr>
</tbody>
</table>

Notice that `Keys.Prior` and `Keys.Next` duplicate the values for `Keys.PageUp` and `Keys.PageDown`. With the exception of Clear, many keyboards duplicate these keys as a separate set of 10 cursor-movement keys that generate the same codes.

I've isolated the following six keys because they also generate character codes and because the `Keys` enumeration values are the same as the character codes they generate:

### Keys Enumeration (ASCII control keys)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>8</td>
</tr>
<tr>
<td>Tab</td>
<td>9</td>
</tr>
<tr>
<td>LineFeed</td>
<td>10</td>
</tr>
<tr>
<td>Enter Return</td>
<td>13</td>
</tr>
<tr>
<td>Escape</td>
<td>27</td>
</tr>
<tr>
<td>Space</td>
<td>32</td>
</tr>
</tbody>
</table>

`Back` is the Backspace key. If present, the numeric keypad provides a second Enter (or Return) key that generates the same code as the normal Enter (or Return) key regardless of the Num Lock state.

The following table shows `Keys` enumeration values for the Shift key, Ctrl key, and Alt key (here called the Menu key because it usually initiates menu selection). Most keyboards these days have pairs of Shift, Ctrl, and Alt keys on the bottom of the keyboard, and the table seems to imply that the left and right versions of these keys generate different codes:

### Keys Enumeration (shift keys)
In reality, however, the enumeration members prefaced with \texttt{L} and \texttt{R} don't appear in any \texttt{KeyEventArgs} object I've ever seen.

These are keys found on the Microsoft Natural Keyboard and clones of that keyboard:

\textbf{Keys Enumeration (Microsoft keys)}

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LWin}</td>
<td>91</td>
<td>Windows flag logo at left</td>
</tr>
<tr>
<td>\texttt{RWin}</td>
<td>92</td>
<td>Windows flag logo at right</td>
</tr>
<tr>
<td>\texttt{Apps}</td>
<td>93</td>
<td>Application menu icon</td>
</tr>
</tbody>
</table>

In this case, the left and right keys \textbf{do} generate different codes.

This table is a collection of some miscellaneous keys and combinations:

\textbf{Keys Enumeration (miscellaneous)}

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Cancel}</td>
<td>3</td>
<td>Pause/Break key when Ctrl is pressed</td>
</tr>
<tr>
<td>\texttt{Pause}</td>
<td>19</td>
<td>Pause/Break key when Ctrl isn't pressed</td>
</tr>
<tr>
<td>\texttt{Capital CapsLock}</td>
<td>20</td>
<td>Caps Lock key</td>
</tr>
<tr>
<td>\texttt{Snapshot PrintScreen}</td>
<td>44</td>
<td>Print Scrn key</td>
</tr>
<tr>
<td>\texttt{NumLock}</td>
<td>144</td>
<td>Num Lock key</td>
</tr>
<tr>
<td>\texttt{Scroll}</td>
<td>145</td>
<td>Scroll Lock key</td>
</tr>
</tbody>
</table>

Five of the \texttt{Keys} enumeration values actually refer to mouse buttons:

\textbf{Keys Enumeration (mouse buttons)}

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{LButton}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{RButton}</td>
<td>2</td>
</tr>
<tr>
<td>\texttt{MButton}</td>
<td>4</td>
</tr>
<tr>
<td>\texttt{XButton1}</td>
<td>5</td>
</tr>
<tr>
<td>\texttt{XButton2}</td>
<td>6</td>
</tr>
</tbody>
</table>

You won't see these members in the \texttt{KeyDown} and \texttt{KeyUp} events. And then there's this group of oddballs:

\textbf{Keys Enumeration (special keys)}

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Select}</td>
<td>41</td>
</tr>
<tr>
<td>\texttt{Print}</td>
<td>42</td>
</tr>
</tbody>
</table>
### Keys Enumeration (special keys)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>43</td>
</tr>
<tr>
<td>Help</td>
<td>47</td>
</tr>
<tr>
<td>ProcessKey</td>
<td>229</td>
</tr>
<tr>
<td>Attn</td>
<td>246</td>
</tr>
<tr>
<td>Crsel</td>
<td>247</td>
</tr>
<tr>
<td>Exsel</td>
<td>248</td>
</tr>
<tr>
<td>EraseEof</td>
<td>249</td>
</tr>
<tr>
<td>Play</td>
<td>250</td>
</tr>
<tr>
<td>Zoom</td>
<td>251</td>
</tr>
<tr>
<td>NoName</td>
<td>252</td>
</tr>
<tr>
<td>Pa1</td>
<td>253</td>
</tr>
<tr>
<td>OemClear</td>
<td>254</td>
</tr>
</tbody>
</table>

If I ever sat down at a keyboard with all these keys, I wouldn't know what to do with them.

The following 12 Keys values apply only to Windows 2000 and later. These keys also generate character codes:

### Keys Enumeration (symbols)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OemSemicolon</td>
<td>186</td>
</tr>
<tr>
<td>Oemplus</td>
<td>187</td>
</tr>
<tr>
<td>Oemcomma</td>
<td>188</td>
</tr>
<tr>
<td>OemMinus</td>
<td>189</td>
</tr>
<tr>
<td>OemPeriod</td>
<td>190</td>
</tr>
<tr>
<td>OemQuestion</td>
<td>191</td>
</tr>
<tr>
<td>Oemtilde</td>
<td>192</td>
</tr>
<tr>
<td>OemOpenBrackets</td>
<td>219</td>
</tr>
<tr>
<td>OemPipe</td>
<td>220</td>
</tr>
<tr>
<td>OemCloseBrackets</td>
<td>221</td>
</tr>
<tr>
<td>OemQuotes</td>
<td>222</td>
</tr>
<tr>
<td>Oem8</td>
<td>223</td>
</tr>
<tr>
<td>OemBackslash</td>
<td>226</td>
</tr>
</tbody>
</table>

For example, the `OemSemicolon` key code is generated when the user presses and releases the key displaying the semicolon and colon.

These key codes are generated for special browser-enabled and media player–enabled keyboards (such as the Microsoft Natural Keyboard Pro or Microsoft Internet Keyboard Pro) in Windows 2000 and later.
These key codes can obviously be ignored by many applications.

The following key codes are generated in connection with the Input Method Editor (IME), which is used to enter ideographs in Chinese, Japanese, and Korean:

### Keys Enumeration (IME)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HangulMode</td>
<td>21</td>
</tr>
<tr>
<td>HangulMode KanaMode</td>
<td></td>
</tr>
<tr>
<td>JunjaMode</td>
<td>23</td>
</tr>
<tr>
<td>FinalMode</td>
<td>24</td>
</tr>
<tr>
<td>KanjiMode HanjaMode</td>
<td>25</td>
</tr>
<tr>
<td>IMEConvert</td>
<td>28</td>
</tr>
<tr>
<td>IMENonconvert</td>
<td>29</td>
</tr>
<tr>
<td>IMEAceet</td>
<td>30</td>
</tr>
<tr>
<td>IMEModeChange</td>
<td>31</td>
</tr>
</tbody>
</table>

Applications are generally only interested in the Unicode character codes that result from the use of the IME.

All the Keys members listed so far have been key codes; that is, they refer to particular keys that are pressed or released. The KeyCode property of the KeyEventArgs object delivered with the KeyDown or KeyUp event will be set to one of the preceding codes.

The Keys enumeration also includes these modifier codes:
Notice that these are bit values. These modifier codes indicate if the Shift, Ctrl, or Alt keys were already pressed when the key-down or key-up event took place. You'll recall in a previous table that I showed key codes for `ShiftKey`, `ControlKey`, and `Menu`. Those key codes indicate the actual Shift, Ctrl, or Alt key being pressed or released.

Three of the read-only properties in `KeyEventArgs`—`KeyCode`, `Modifiers`, and `KeyData`—are all of type `Keys`. Each key pressed or released generates one event:

- The `KeyCode` property indicates the key being pressed or released. These keys can include the Shift (indicated by `Keys.ShiftKey`), Ctrl (`Keys.ControlKey`), or Alt key (`Keys.Menu`).
- The `Modifiers` property indicates the state of the Shift, Ctrl, and Alt keys during the key press or release. `Modifiers` can be any combination of `Keys.Shift`, `Keys.Control`, or `Keys.Alt`. Or if no modifier key is pressed, `Modifiers` is `Keys.None`, which is defined simply as 0.
- The `KeyData` property is a combination of `KeyCode` and `Modifiers`.

For example, let's assume you press the Shift key and then D, and then release D and release Shift. This table shows the four events and the `KeyEventArgs` properties associated with these key actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Event</th>
<th>KeyCode</th>
<th>Modifiers</th>
<th>KeyData</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release Shift</td>
<td>KeyUp</td>
<td><code>Keys.ShiftKey</code></td>
<td><code>Keys.None</code></td>
<td><code>Keys.ShiftKey</code></td>
</tr>
</tbody>
</table>

If you're working with the `KeyData` property, the `Keys` enumeration also defines two masks to differentiate the key codes and the modifiers:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyCode</td>
<td>0x0000FFFF</td>
<td>Bit mask for key codes</td>
</tr>
<tr>
<td>Modifiers</td>
<td>0xFFFF0000</td>
<td>Bit mask for modifier keys</td>
</tr>
</tbody>
</table>

Notice that these enumeration members have the same names as the corresponding properties of the `KeyEventArgs` class. If the `KeyEventArgs` object is named `kea`, the expression

`kea.KeyData & Keys.KeyCode`

is the same as `kea.KeyCode` and the expression

`key.KeyData & Keys.Modifiers`

is the same as `kea.Modifiers`.

**Testing the Modifier Keys**
It's not necessary to be handling a KeyDown or KeyUp event to determine whether the Shift, Ctrl, or Alt key is pressed. You can also obtain the current state of the three modifier keys using the static Control.ModifierKeys property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keys</td>
<td>ModifierKeys</td>
<td>get</td>
<td>State of the Shift, Alt, and Ctrl keys</td>
</tr>
</tbody>
</table>

Suppose you needed to do something different depending on whether the Shift or Ctrl key—or both—were pressed, but not if the Alt key were pressed. You would first call the static ModifierKeys property:

```csharp
Keys keysMod = Control.ModifierKeys;
```

You then test the possible combinations you're interested in with code that looks like this:

```csharp
if (keysMod == (Keys.Shift | Keys.Control))
{
    // Shift and Ctrl keys are pressed.
}
else if (keysMod == Keys.Shift)
{
    // Shift key is pressed.
}
else if (keysMod == Keys.Control)
{
    // Ctrl key is pressed.
}
```

You might need to use ModifierKeys when you're not processing a KeyDown or KeyUp event, perhaps when you're processing a mouse event. We've all seen programs that interpret mouse clicks and mouse movement differently when the Shift or Ctrl key is pressed. This is the kind of situation in which the ModifierKeys property is useful. An example is the CanonicalSpline program in Chapter 13.

Unfortunately, there is no way to test the state of the toggle keys Caps Lock, Num Lock, and Scroll Lock.

**Reality Check**

Although a Windows Forms program certainly gets a lot of information about keystrokes, most programs can ignore most of them. If you process the KeyDown event, for example, you usually don't have to bother with the KeyUp event.

Windows Forms programs often ignore events involving keystrokes that also generate characters. (I'll get to the keyboard character event shortly.) You might have concluded that you can get all the keyboard information you need from the KeyDown and KeyUp events to generate your own character codes.

This is a bad idea. For example, suppose your event handler gets a KeyEventArg object with a KeyCode property of Keys.D3 and a Modifiers property of Keys.Shift. You know what that is, right? It's the pound sign (#), which has an ASCII and Unicode encoding of 0x0023. Well, maybe. In the United Kingdom, the upper-shift 3 key generates another type of pound sign, which has a character encoding of 0x00A3 and looks like this: £.

A more serious problem involves the Caps Lock key. As I mentioned earlier, there is no facility in Windows Forms to test the state of Caps Lock. You can tell when Caps Lock is being pressed and released, but Caps Lock could already be toggled on when your program begins executing.
The **KeyDown** event is most useful for the cursor-movement keys, the function keys, Insert, and Delete. However, the function keys often appear as menu accelerators. Because menu accelerator keys are translated into menu command events automatically, you don't have to process the keystrokes themselves. Function keys, too, often duplicate the functionality of menu items. And when programs define function keys that don't duplicate menu items—when function keys are used in combination with Shift and Ctrl with the crazy abandon of old MS-DOS programs such as WordPerfect and Microsoft Word—then those programs aren't being very user friendly.

So it comes down to this: most of the time you'll process **KeyDown** events only for cursor-movement keys, Insert, and Delete. When you use those keys, you can check the shift state with the **Modifiers** property of the **KeyEventArgs** object. Programs often use the Shift key in combination with the cursor keys to extend a selection in (for example) a word processing document. The Ctrl key is often used to alter the meaning of the cursor keys. For example, Ctrl in combination with the right arrow key might mean to move the cursor one **word** to the right rather than one character.

### A Keyboard Interface for SysInfo

I assume you recall the various programs from Chapter 4 that displayed system information. The last one was SysInfoReflection, and it had progressed a great deal from the earliest tentative code. But it still had one little problem: it had no keyboard interface.

The time has come to add one, and here's another example in which inheritance really pays off. This class derives from the **SysInfoReflection** class and adds an override of the **OnKeyDown** method.

**SysInfoKeyboard.cs**

```csharp
//SysInfoKeyboard.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoKeyboard: SysInfoReflection
{
    public new static void Main()
    {
        Application.Run(new SysInfoKeyboard());
    }
    public SysInfoKeyboard()
    {
        Text = "System Information: Keyboard";
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        Point pt = AutoScrollPosition;

        pt.X = -pt.X;

        switch(kea.KeyCode)
        {
            case Keys.Right:
                if ((kea.Modifiers & Keys.Control) == Keys.Control)
```
As I said in Chapter 4, the read/write AutoScrollPosition property is implemented in ScrollableControl (of which Form is a descendent) as part of the support for auto-scroll. AutoScrollPosition is a Point structure that indicates the positions of the two scroll bars.

When you get the AutoScrollPosition value, the coordinates are negative, indicating the location of the virtual client area relative to the upper left corner of the physical client area. When you set AutoScrollPosition, however, the coordinates must be positive. That's the reason for the two lines of code:

```csharp
pt.X = -pt.X;
```

Otherwise, the coordinates are simply adjusted based on the particular cursor key. For the left and right arrow keys, I shift the client area by the width of the client area if the Ctrl key is pressed and by the height of a Font character if not. I've made the effects of other cursor keys independent of any modifier keys. The Home key returns the display back to the origin; the End key goes to the bottom of the list but doesn't change the horizontal position.

**KeyPress for Characters**

Many keys on the keyboard generate character codes. To get those codes, you install an event handler for KeyPress or (if possible) override the OnKeyPress method:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyPress</td>
<td>OnKeyPress</td>
<td>KeyPressEventHandler</td>
<td>KeyPressEventArgs</td>
</tr>
</tbody>
</table>
The `KeyPressEventArgs` class has just two properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>KeyChar</td>
<td>get</td>
<td>Unicode character code</td>
</tr>
<tr>
<td>bool</td>
<td>Handled</td>
<td>get/set</td>
<td>Set by event handler (initially false)</td>
</tr>
</tbody>
</table>

The `char` data type is, of course, a 16-bit Unicode character.

Refer to the table presented earlier (page 226) that showed the events corresponding to pressing the Shift key and the D key. Right in the middle of this process—between the pair of `KeyDown` events and the pair of `KeyUp` events—you'll get a `KeyPress` event with a `KeyChar` property of 0x0044, which is the uppercase D. (Well, probably. If Caps Lock is toggled on, you'll get 0x0064, a lowercase d.)

Of course, I'm assuming that you have an American English keyboard layout installed. If you have a Greek keyboard layout installed, you'll get 0x0394, which corresponds to . If you have a Russian keyboard layout installed, you'll get a code of 0x0412, which corresponds to Â. I'll explain how to install foreign keyboard layouts later in this chapter.

**Control Characters**

With the Ctrl key down, you can generate control characters that are reported through the `KeyPress` event. You get character codes 0x0001 through 0x001A by using the Ctrl key in combination with A through Z regardless of the Shift key status. Here are some other control characters you can generate from the keyboard.

<table>
<thead>
<tr>
<th>Key</th>
<th>Control Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift+Ctrl @</td>
<td>0x0000</td>
</tr>
<tr>
<td>Backspace</td>
<td>0x0008</td>
</tr>
<tr>
<td>Tab</td>
<td>0x0009</td>
</tr>
<tr>
<td>Ctrl+Enter</td>
<td>0x000A</td>
</tr>
<tr>
<td>Enter</td>
<td>0x000D</td>
</tr>
<tr>
<td>Esc</td>
<td>0x001B</td>
</tr>
<tr>
<td>Ctrl [</td>
<td></td>
</tr>
<tr>
<td>Ctrl \</td>
<td>0x001C</td>
</tr>
<tr>
<td>Ctrl ]</td>
<td>0x001D</td>
</tr>
<tr>
<td>Shift+Ctrl ^</td>
<td>0x001E</td>
</tr>
<tr>
<td>Shift+Ctrl _</td>
<td>0x001F</td>
</tr>
<tr>
<td>Ctrl+Backspace</td>
<td>0x007F</td>
</tr>
</tbody>
</table>

Programs often use Shift in combination with Tab to tab backward. There's no special code for that; it's something you'll have to handle on your own.

A bit of overlap occurs between the `Keys` enumeration codes you get with the `KeyDown` and `KeyUp` events and the character codes you get with the `KeyPress` event. Which should you use to process the Tab key, Enter, Backspace, or Esc?

I've gone back and forth on this issue for the past 15 years, but these days I prefer treating these keys as Unicode control characters rather than keystrokes. The only good reason I can come up with, however, is that some old-time users may type Ctrl+H for Backspace or Ctrl+I for Tab, and you want to make sure those key combinations work correctly. You get that functionality by processing Backspace and Tab in the `KeyPress` event handler.
Looking at the Keys

You may be wondering where I got all the information I've been telling you about what you'll see in
the keyboard events since it certainly isn't in the Windows Forms documentation. Well, most of what
I know about the keyboard was revealed by the following program, which displays information about
keys as you type them and which keeps the last 25 keyboard events (KeyDown, KeyUp, and
KeyPress) stored in an array.

**KeyExamine.cs**

```csharp
// KeyExamine.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

public class KeyExamine : Form
{
    public static void Main()
    {
        Application.Run(new KeyExamine());
    }

    enum EventType
    {
        None,
        KeyDown,
        KeyUp,
        KeyPress
    }

    struct KeyEvent
    {
        public EventType evttype;
        public EventArgs evtargs;
    }

    const int iNumLines = 25;
    int iNumValid = 0;
    int iInsertIndex = 0;
    KeyEvent[] akeyevt = new KeyEvent[iNumLines];

    // Text positioning
    int xEvent, xChar, xCode, xMods, xData,
        xShift, xCtrl, xAlt, xRight;
```
public KeyExamine()
{
    Text = "Key Examine";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    xEvent = 0;
xChar = xEvent + 5 * Font.Height;
xCode = xChar + 5 * Font.Height;
xMods = xCode + 8 * Font.Height;
xData = xMods + 8 * Font.Height;
xShift = xData + 8 * Font.Height;
xCtrl = xShift + 5 * Font.Height;
xAlt = xCtrl + 5 * Font.Height;
xRight = xAlt + 5 * Font.Height;

    ClientSize = new Size(xRight, Font.Height * (iNumLines + 1));
    FormBorderStyle = FormBorderStyle.Fixed3D;
    MaximizeBox = false;
}

protected override void OnKeyDown(KeyEventArgs kea)
{
    akeyevt[iInsertIndex].evttype = EventType.KeyDown;
    akeyevt[iInsertIndex].evtargs = kea;
    OnKey();
}

protected override void OnKeyUp(KeyEventArgs kea)
{
    akeyevt[iInsertIndex].evttype = EventType.KeyUp;
    akeyevt[iInsertIndex].evtargs = kea;
    OnKey();
}

protected override void OnKeyPress(KeyPressEventArgs kpea)
{
    akeyevt[iInsertIndex].evttype = EventType.KeyPress;
    akeyevt[iInsertIndex].evtargs = kpea;
    OnKey();
}

void OnKey()
{
    if(iNumValid < iNumLines)
    {
        Graphics grfx = CreateGraphics();
        DisplayKeyInfo(grfx, iInsertIndex, iInsertIndex);
        grfx.Dispose();
    }
else
{
    ScrollLines();
}
iInsertIndex = (iInsertIndex + 1) % iNumLines;
iNumValid = Math.Min(iNumValid + 1, iNumLines);
}
protected virtual void ScrollLines()
{
    Rectangle rect = new Rectangle(0, Font.Height,
        ClientSize.Width,
        ClientSize.Height - Font.Height);

    // I wish I could scroll here!

    Invalidate(rect);
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    BoldUnderline(grfx, "Event", xEvent, 0);
    BoldUnderline(grfx, "KeyChar", xChar, 0);
    BoldUnderline(grfx, "KeyCode", xCode, 0);
    BoldUnderline(grfx, "Modifiers", xMods, 0);
    BoldUnderline(grfx, "KeyData", xData, 0);
    BoldUnderline(grfx, "Shift", xShift, 0);
    BoldUnderline(grfx, "Control", xCtrl, 0);
    BoldUnderline(grfx, "Alt", xAlt, 0);

    if(iNumValid < iNumLines)
    {
        for (int i = 0; i < iNumValid; i++)
            DisplayKeyInfo(grfx, i, i);
    }
    else
    {
        for (int i = 0; i < iNumLines; i++)
            DisplayKeyInfo(grfx, i, (iInsertIndex + i) %
                iNumLines);
    }
}
void BoldUnderline(Graphics grfx, string str, int x, int y)
{
    // Draw the text bold.
Brush brush = new SolidBrush(ForeColor);
grfx.DrawString(str, Font, brush, x, y);
grfx.DrawString(str, Font, brush, x + 1, y);

    // Underline the text.

SizeF sizef = grfx.MeasureString(str, Font);
grfx.DrawLine(new Pen(ForeColor), x, y + sizef.Height,
    x + sizef.Width, y + sizef.Height);
}
void DisplayKeyInfo(Graphics grfx, int y, int i)
{
    Brush br = new SolidBrush(ForeColor);
y = (1 + y) * Font.Height;    // Convert y to pixel coordinate.
grfx.DrawString(akeyevt[i].evttype.ToString(),
    Font, brush, xEvent, y);

    if(akeyevt[i].evttype == EventType.KeyPress)
    {
        KeyPressEventArgs kpea =
            (KeyPressEventArgs) akeyevt[i].evtargs;

        string str = String.Format("\x202D{0} (0x{1:X4})",
            kpea.KeyChar, (int) kpea.KeyChar);
grfx.DrawString(str, Font, br, xChar, y);
    }
    else
    {
        KeyEventArgs kea = (KeyEventArgs) akeyevt[i].evtargs;

        string str = String.Format("{0} {{1}}",
            kea.KeyCode, (int) kea.KeyCode);
grfx.DrawString(str, Font, br, xCode, y);
grfx.DrawString(kea.Modifiers.ToString(), Font, br, xMods, y);
grfx.DrawString(kea.KeyData.ToString(), Font, br, xData, y);
grfx.DrawString(kea.Shift.ToString(), Font, br, xShift, y);
grfx.DrawString(kea.Control.ToString(), Font, br, xCtrl, y);
grfx.DrawString(kea.Alt.ToString(), Font, br, xAlt, y);
    }
}
This is a fairly large program for this book. Early in the class is the definition of a private enumeration (named EventType) and a structure (named KeyEvent) used for storing the KeyEventArgs and KeyPressEventArgs information associated with each keystroke. The program then creates an array of 25 of these structures. The integer fields beginning with the prefix x are used for positioning the information into columns.

As each KeyDown, KeyUp, and KeyPress event comes through, the event information is stored in the array and also displayed on the client area by the method named DisplayKeyInfo, which is the largest method in the KeyExamine class. The OnPaint method also makes use of the DisplayKeyInfo method and displays column headers bolded and underlined. I'll present a much better way of getting a bold underlined font in Chapter 9; this program simply draws the text twice, the second time offset from the first by a pixel, and then uses DrawLine to draw a line underneath the text. Here's the program after typing "Hello!":

One problem with this program is that when it gets down to the bottom of the client area, it wants to scroll everything up. If I were writing Win32 code, I'd use the ScrollWindow call for doing that. However, nothing like that function is available in Windows Forms. So instead of scrolling, the program simply invalidates that part of the client area below the headings, forcing the OnPaint method to repaint all the lines. It doesn't really work very well and I feel awful doing it, but probably not as bad as the person at Microsoft who forgot to implement ScrollWindow in Windows Forms!

**Invoking the Win32 API**

So, what do you do if you really, really, really need to use a Win32 API function and it's simply not available in the .NET Framework?

If necessary, you can resort to using Platform Invocation Services. PInvoke (as it's called) is a generalized mechanism that allows you to call functions exported from DLLs. The ScrollWindow function happens to be located on your machine in the dynamic-link library User32.dll, so that certainly qualifies. The drawback is that a programmer who uses this facility is no longer writing managed code, and certainly not platform-independent code.

The documentation for the Win32 API shows the following syntax for ScrollWindow:

```c
BOOL ScrollWindow(HWND hWnd, int XAmount, int YAmount, CONST RECT *lpRect, CONST RECT *lpClipRect);
```

In the C header files for Windows, BOOL is simply defined as an int data type, and the HWND (a handle to a window) is defined as a pointer to void, but it's really just a 32-bit value.

Where are we going to get a window handle in Windows Forms? Well, the Control class has a Handle property, which is documented as the control's HWND. The type of the Handle property is an IntPtr structure, which is defined in the System namespace and indicates a pointer. You can easily convert between the int and IntPtr data types. So far, we have a fairly clean transition between C# data types and the arguments and return type of the ScrollWindow call.
The tough part involves the last two arguments to ScrollWindow. These arguments are pointers to Windows RECT structures. The RECT structure is defined in a Windows header file like so:

typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;

The LONG data type is defined in a Windows header file as a long, but that’s not a 64-bit C# long; it’s only a 32-bit C long, so it too is compatible with the C# int.

To call ScrollWindow from a C# program, you must define a struct that has the same fields in the same order as the Windows RECT structure and preface it with the attribute
[StructLayout(LayoutKind.Sequential)]

StructLayout is a C# attribute based on the StructLayoutAttribute class defined in the System.Runtime.InteropServices namespace. You must also declare ScrollWindow as an extern function and preface it with the attribute
[DllImport("user32.dll")]

You may have noticed that the KeyExamine class contains a method I called ScrollLines that is responsible for scrolling the contents of the client area. The ScrollLines method in KeyExamine simply invalidated that portion of the client area below the titles. Here’s a class that inherits from KeyExamine, defines a RECT structure, declares the ScrollWindow function, and overrides the ScrollLines method in KeyExamine. This revised version of ScrollLines calls the Windows ScrollWindow function.

KeyExamineWithScroll.cs
//---------------------------------------------------
// KeyExamineWithScroll.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Runtime.InteropServices;
using System.Windows.Forms;

class KeyExamineWithScroll: KeyExamine
{
    public new static void Main()
    {
        Application.Run(new KeyExamineWithScroll());
    }
    public KeyExamineWithScroll()
    {
        Text += " With Scroll";
    }
    // Define a Win32-like rectangle structure.
    [StructLayout(LayoutKind.Sequential)]
    public struct RECT
This version is much better: the program runs smoother and more efficiently when it doesn't have to redraw all the lines of output.

**Handling Input from Foreign Keyboards**

It's always helpful to test your programs on any type of keyboard that's not like the one on your machine, and in particular, foreign language keyboards. And running your program with Russian is much easier than a trip to Moscow. You can install foreign-language keyboard layouts, which are small files that let you use your existing keyboard to generate character codes from other languages.

The following instructions for installing foreign-language keyboard layouts pertain to the English version of Windows 2000.

In Control Panel, run Regional Options. Select the General tab. Where it says Language Settings For The System, probably only Western Europe And United States (the default) is checked. Check at least Arabic, Cyrillic, Greek, and Hebrew, and click OK. You'll need to have your Windows 2000 CD-ROM handy, and the system will want to reboot itself.

After you've rebooted, bring up Control Panel again and run Keyboard. Select the Input Locales tab. In the Installed Input Locales section, click the Add button, and, one by one, add Input Locales of the following if they're not already installed: Arabic (Egypt), English (United Kingdom), English (United States), German (Germany), Greek, Hebrew, and Russian. This process will also install keyboard layouts associated with these input locales.

If you've never added additional keyboard layouts and you left the Enable Indicator On Taskbar option in the Keyboard Properties dialog box checked, you'll see a new icon in the tray section of
your taskbar: a box with the letters EN (meaning English). You can click on that icon and switch to an alternative English keyboard or to an Arabic, a German, a Greek, a Hebrew, or a Russian one. This change affects only the currently active application.

Now let's experiment a bit with KeyExamine or KeyExamineWithScroll. Run one of these programs and switch to the English (United States) keyboard layout if it's not set for that already. Type an upper-shift 3. You'll get a KeyChar code of 0x0023 and a # character. Switch to English (United Kingdom) and type the same key combination. Now it's a code of 0x00A3 and a £ character.

Switch to the German (Germany) keyboard layout. Type a Y and a Z. Notice that both the KeyCode and KeyChar codes indicate a Z when you type Y and a Y when you type Z. That's because these two characters are switched around on the German keyboard.

While still running the German (Germany) keyboard layout, press the +/= key. The KeyCode is 221, which corresponds to Keys.OemCloseBrackets. Now type the A key. The result is a lowercase a with an acute accent: á, Unicode character 0x00E1. The +/= key on the German keyboard is known as a dead key. You follow a dead key with an appropriate character key and the result is an accented character key. You can follow the +/= key with any uppercase or lowercase vowel: a, e, i, o, u, or y (which is actually produced by your Z key). The uppercase +/= followed by an uppercase or lowercase vowel (a, e, i, o, or u, but not a y in this case) results in that letter with a grave accent, for example à.

If you type a consonant after a dead key, you'll get the accent by itself (an ´ or a ') followed by the letter. To type one of these accents by itself, follow the dead key by the spacebar.

Similarly, on the German keyboard, pressing the ~/' key followed by a, e, i, o, or u results in the letter with a circumflex: â. (The shifted ~/' key isn't a dead key; it generates a ° character.) The umlaut in German appears only on uppercase or lowercase ä, ö, or ü. You can generate these characters by typing the ~/' key, /; key, or /{[ key, respectively.

So far, all the KeyChar values that have accompanied the KeyPress events we've generated have been in the 8-bit range. These are characters that are defined by one of two standards. The first standard is known as ANSI X3.4-1986, "Coded Character Sets—7-Bit American National Standard Code for Information Interchange (7-Bit ASCII)":

<table>
<thead>
<tr>
<th>-0</th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
<th>-4</th>
<th>-5</th>
<th>-6</th>
<th>-7</th>
<th>-8</th>
<th>-9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-</td>
<td>1</td>
<td>*</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>(</td>
<td>)</td>
<td>+</td>
<td>.</td>
<td>-</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
</tr>
<tr>
<td>4-</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>5-</td>
<td>P</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
</tr>
<tr>
<td>6-</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>7-</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The second standard is the ASCII extension documented by ANSI/ISO 8859-1-1987, "American National Standard for Information Processing—8-Bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1" and commonly referred to as "Latin 1":

<table>
<thead>
<tr>
<th>-0</th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
<th>-4</th>
<th>-5</th>
<th>-6</th>
<th>-7</th>
<th>-8</th>
<th>-9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-</td>
<td>i</td>
<td>e</td>
<td>o</td>
<td>v</td>
<td>;</td>
<td>_</td>
<td>@</td>
<td>*</td>
<td>-</td>
<td>_</td>
<td>-</td>
<td>_</td>
<td>-</td>
<td>_</td>
<td>-</td>
</tr>
<tr>
<td>B-</td>
<td>z</td>
<td>s</td>
<td>u</td>
<td>l</td>
<td>.</td>
<td>,</td>
<td>%</td>
<td>½</td>
<td>¼</td>
<td>%</td>
<td>½</td>
<td>¼</td>
<td>%</td>
<td>½</td>
<td>¼</td>
</tr>
<tr>
<td>C-</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>AE</td>
<td>É</td>
<td>É</td>
<td>É</td>
<td>Ê</td>
<td>Ê</td>
<td>Ê</td>
<td>Ê</td>
<td>Ê</td>
</tr>
<tr>
<td>D-</td>
<td>É</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
<td>Ë</td>
</tr>
<tr>
<td>E-</td>
<td>á</td>
<td>á</td>
<td>á</td>
<td>á</td>
<td>á</td>
<td>á</td>
<td>è</td>
<td>é</td>
<td>é</td>
<td>é</td>
<td>é</td>
<td>é</td>
<td>é</td>
<td>é</td>
<td>é</td>
</tr>
<tr>
<td>F-</td>
<td>ö</td>
<td>ö</td>
<td>ö</td>
<td>ö</td>
<td>ö</td>
<td>ö</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
<td>ë</td>
</tr>
</tbody>
</table>

These character sets are suitable only for languages that use the Latin alphabet. To accommodate other alphabets of the world (as well as the ideographs of Chinese, Japanese, and Korean), the 16-bit character encoding known as Unicode was developed. Windows Forms programs written in C#
generally don't need to do anything special to support Unicode. The char data type in C# is 16 bits wide, for example.

If you switch to the Russian keyboard layout and type a few keys, you'll see Cyrillic letters. These have character codes in the range from 0x0410 through 0x044F, which is defined in the Unicode standard as the Basic Russian Alphabet. Similarly, you can switch to the Arabic, Greek, or Hebrew keyboard layout and type letters in those alphabets.

If you've never explored this stuff before, you may be wondering how foreign-language alphabets and keyboards worked before Unicode—when character codes were just 8 bits wide. Well, in short, it was a mess.

If you'd like your program to be informed when the user changes the keyboard layout, you can install event handlers for the InputLanguageChanging and InputLanguageChanged events or override the OnInputLanguageChanging and OnInputLanguageChanged methods. In the following table, ellipses are used to indicate the event name in the method, delegate, and event argument names:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputLanguageChanging</td>
<td>On…</td>
<td>…EventHandler</td>
<td>…EventArgs</td>
</tr>
<tr>
<td>InputLanguageChanged</td>
<td>On…</td>
<td>…EventHandler</td>
<td>…EventArgs</td>
</tr>
</tbody>
</table>

You get the InputLanguageChanging event first. The InputLanguageChangingEventArgs object has information about the language the user wants to switch to. If it's not OK with your program to make this switch, set the Cancel property of the InputLanguageChangingEventArgs object to true; otherwise, you'll soon receive an InputLanguageChanged event.

To pursue this subject further, take a look at the InputLanguage class in the System.Windows.Forms namespace and the CultureInfo class in the System.Globalization namespace. [Diagrams of many foreign-language keyboards are in Nadine Kano's Developing International Software for Windows 95 and Windows NT. This Microsoft Press book is out of print, but an electronic version is available on the MSDN library Web site (http://msdn.microsoft.com/library, under Development (General) and Internationalization).]

**Input Focus**

Input focus is an important issue when you begin creating controls on the surface of your form. Input focus determines which control gets keyboard input. In a dialog box, some keys (such as Tab and the cursor-movement keys) shift input focus among the controls.

*Form* inherits three read-only properties that pertain to input focus:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>CanFocus</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>ContainsFocus</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>Focused</td>
<td>get</td>
</tr>
</tbody>
</table>

A control (or form) can't get the input focus if it is disabled or invisible. You can use the CanFocus property to check this state. The ContainsFocus property returns true if the control (or form) or one of its children has the input focus. Focused returns true if the control (or form) has the input focus.

A program can set the input focus to one of its controls by using the Focus method.
The return value indicates whether focus was successfully applied. It won't succeed if the control isn't a child of the active form.

Finally, two events tell a control (or form) when it is getting input focus and when it is losing input focus:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>GotFocus</td>
<td>OnGotFocus</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>LostFocus</td>
<td>OnLostFocus</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

A control (or form) always eventually gets a *LostFocus* event to match every *GotFocus* event. I'll have more to say about input focus when we begin creating controls in Chapter 12.

**The Missing Caret**

Controls or forms that accept keyboard input generally display something special when they have input focus. A button control, for example, displays a dotted outline around its text. Controls or forms that allow you to type text usually display a little underline, a vertical bar, or a box that shows you where the next character you type will appear on the screen. You may know this indicator as a *cursor*, but in Windows it's more properly known as a *caret*. The word *cursor* is reserved for the bitmap picture representing the mouse position.

If you create a *TextBox* or a *RichTextBox* control (which I'll demonstrate in Chapter 18), the control is responsible for creating and displaying the caret. In many cases, using these controls will serve your program well. *RichTextBox* in particular is quite powerful and is built around the same Windows control that Microsoft WordPad uses.

However, if these controls are not adequate for your purposes and you need to write your own text-input code, you have a little problem. Of the several features missing from the Windows Forms class libraries, perhaps none is more inexplicable than the caret.

I'm afraid it's time again to create some unmanaged code that digs into the Windows DLLs to do what we need to do. My class named *Caret* is defined in my own personalized namespace in case you want to use it in one of your own programs. It's based on the Windows caret API and begins by declaring five external functions located in User32.dll.

**Caret.cs**

```csharp
using System;
using System.Drawing;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsWithCSharp
{
    class Caret
    {
        [DllImport("user32.dll")] public static extern int CreateCaret(IntPtr hwnd, IntPtr hbm, int cx, int cy);
        [DllImport("user32.dll")] public static extern int DestroyCaret();
    }
}
```
public static extern int SetCaretPos(int x, int y);

public static extern int ShowCaret(IntPtr hwnd);

public static extern int HideCaret(IntPtr hwnd);

// Fields
Control ctrl;
Size    size;
Point   ptPos;
bool    bVisible;

// Constructors

// Don't allow default constructor.
private Caret()
{
}

// Only allowable constructor has Control argument.

public Caret(Control ctrl)
{
    this.ctrl = ctrl;
    Position = Point.Empty;
    Size     = new Size(1, ctrl.Font.Height);

    Control.GotFocus += new EventHandler(ControlOnGotFocus);
    Control.LostFocus += new EventHandler(ControlOnLostFocus);

    // If the control already has focus, create the caret.

    if (ctrl.Focused)
        ControlOnGotFocus(ctrl, new EventArgs());
}

// Properties

public Control Control
{
    get
    {
        return ctrl;
    }
}

public Size Size
{
    get
{    return size; }
}
set
{    size = value; }
}

public Point Position
{
    get
    {
        return ptPos;
    }
    set
    {
        ptPos = value;
        SetCaretPos(ptPos.X, ptPos.Y);
    }
}

public bool Visibility
{
    get
    {
        return bVisible;
    }
    set
    {
        if (bVisible = value)
            ShowCaret(Control.Handle);
        else
            HideCaret(Control.Handle);
    }
}

public void Show()
{
    Visibility = true;
}

public void Hide()
{
    Visibility = false;
}

public void Dispose()
{
// If the control has focus, destroy the caret.
if (ctrl.Focused)
    ControlOnLostFocus(ctrl, new EventArgs());

Control.GotFocus -= new EventHandler(ControlOnGotFocus);  
Control.LostFocus -= new EventHandler(ControlOnLostFocus);

// Event handlers
void ControlOnGotFocus(object obj, EventArgs ea)
{
    CreateCaret(Control.Handle, IntPtr.Zero,  
        Size.Width, Size.Height);  
    SetCaretPos(Position.X, Position.Y);  
    Show();
}

void ControlOnLostFocus(object obj, EventArgs ea)
{
    Hide();
    DestroyCaret();
}

To create a caret for your form (or any other object derived from Control), use the constructor
Caret caret = new Caret(form);

The Caret class defines the default constructor as private, so you must include an argument in the
constructor. Caret provides four properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Control</td>
<td>get</td>
<td>Control object the caret is associated with</td>
</tr>
<tr>
<td>Size</td>
<td>Size</td>
<td>get/set</td>
<td>Size of caret in pixels</td>
</tr>
<tr>
<td>Point</td>
<td>Position</td>
<td>get/set</td>
<td>Position of caret relative to control origin</td>
</tr>
<tr>
<td>bool</td>
<td>Visibility</td>
<td>get/set</td>
<td>Visibility of caret</td>
</tr>
</tbody>
</table>

In character mode environments, carets are often underlines or boxes. These shapes don’t quite
work right for variable-width text, however; a vertical line is better. Generally, a program that uses the
Caret class in connection with the default font for the control will set the size like so:
caret.Size = new Size(2, Font.Height);

The Position property indicates the position of the caret relative to the upper left corner of the client
area.

You can use the Visibility property to hide and reshow the caret. You must hide the caret when you
draw on your form at any time other than during the Paint event! As an alternative to Visibility, you
can use the Hide and Show methods. The Dispose method is the only other public method Caret supports:
### Caret Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Hide()</td>
<td>Hides the caret</td>
</tr>
<tr>
<td>void Show()</td>
<td>Shows the caret</td>
</tr>
<tr>
<td>void Dispose()</td>
<td>Disables the caret</td>
</tr>
</tbody>
</table>

Normally, you don't need to call `Dispose`. The only time `Dispose` is necessary is if you've been using the caret to perform some keyboard input in your form or control and you no longer need to do so.

The `Caret` class is a good example of a class that must install event handlers for the form that it's associated with. `Caret` installs event handlers for the `GotFocus` and `LostFocus` events; it creates the caret when the form gets the focus and destroys the caret when the form loses the focus. This approach is in accordance with recommended handling of the caret in Win32 programming. `Dispose` simply uninstalls the event handlers so the caret isn't created anymore.

But keep this in mind: A form that uses this `Caret` class and that itself overrides its `OnGotFocus` and `OnLostFocus` methods runs the risk of disabling the event handlers in `Caret`. If you need to override these methods, be sure to call the method in the base class:

```csharp
protected override void OnGotFocus(EventArgs ea)
{
    base.OnGotFocus(ea);
    ;
}
protected override void OnLostFocus(EventArgs ea)
{
    base.OnLostFocus(ea);
    ;
}
```

Those base class `OnGotFocus` and `OnLostFocus` methods call the installed event handlers such as the ones in `Caret`.

### Echoing Key Characters

Now let's look at a program that uses the `Caret` class to let you enter and edit text. This program comes very close to the functionality of a `TextBox` control in single-line mode.

**TypeAway.cs**

```csharp
// TypeAway.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;
using Petzold.ProgrammingWindowsWithCSharp;

class TypeAway: Form
{
    public static void Main()
    {
        Application.Run(new TypeAway());
```
protected Caret caret;
protected string strText = "";
protected int iInsert = 0;

public TypeAway()
{
    Text = "Type Away";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    FontHeight = 24;

    caret = new Caret(this);
    caret.Size = new Size(2, Font.Height);
    caret.Position = new Point(0, 0);
}

protected override void OnKeyPress(KeyPressEventArgs kpea)
{
    caret.Hide();
    Graphics grfx = CreateGraphics();
    grfx.FillRectangle(new SolidBrush(BackColor),
        new RectangleF(Point.Empty,
        grfx.MeasureString(strText, Font,
        Point.Empty,
        StringFormat.GenericTypographic)));
    switch(kpea.KeyChar)
    {
        case '\b':
            if (iInsert > 0)
            {
                strText = strText.Substring(0, iInsert - 1) +
                strText.Substring(iInsert);
                iInsert -= 1;
            }
            break;

        case '\r':
            case '\n':
                break;

        default:
            if (iInsert == strText.Length)
                strText += kpea.KeyChar;
            break;
    }
}
else
    strText = strText.Substring(0, iInsert) +
    kpea.KeyChar +
    strText.Substring(iInsert);
    iInsert++;
    break;
}
gfx.TextRenderingHint = TextRenderingHint.AntiAlias;
gfx.DrawString(strText, Font, new SolidBrush(ForeColor),
    0, 0, StringFormat.GenericTypographic);
gfx.Dispose();

    PositionCaret();
    caret.Show();
}
protected override void OnKeyDown(KeyEventArgs kea)
{
    switch (kea.KeyData)
    {
    case Keys.Left:
        if (iInsert > 0)
            iInsert--;
        break;

    case Keys.Right:
        if (iInsert < strText.Length)
            iInsert++;
        break;

    case Keys.Home:
        iInsert = 0;
        break;

    case Keys.End:
        iInsert = strText.Length;
        break;

    case Keys.Delete:
        if (iInsert < strText.Length)
        {
            iInsert++;
            OnKeyPress(new KeyPressEventArgs('\b'));
        }
        break;
The `TypeAway` class creates an object of type `Caret` in its constructor and initializes the size and position. The program needs only to hide and then reshow the caret when drawing on the form at times other than the `Paint` event and to set the caret's position within the client area.

The string of characters that the user enters and edits is stored in the field named `strText`. The `iInsert` field is the insertion point in this string. For example, after you type three characters, `iInsert` equals 3. If you then press the left arrow, `iInsert` equals 2. The `PositionCaret` method in `TypeAway` is responsible for converting that character position to a pixel position that it uses to set the `Position` property of the `Caret` object.

Let's take a look at how `TypeAway` handles the `OnKeyPress` method. In most cases, it may seem that the program simply needs to display this new character on the form at the pixel position corresponding to the current insertion point and to append this character to the `strText` field. Instead, however, the program uses `MeasureString` and `FillRectangle` to entirely erase any text currently displayed in the form! This behavior may sound a little extreme, but (as we'll see) it's necessary if the insertion point isn't at the end of the string or if it's displaying text in some non-English languages.

The `OnKeyPress` method handles the Backspace key by removing a character from the string in front of the insertion point. The method ignores carriage returns and line feeds, and handles all other characters by inserting them into `strText` at the insertion point. The method then displays the entire string and calls `PositionCaret` (which I'll describe shortly). Notice that the method hides the caret while drawing on the form.

The `OnKeyDown` method handles a few cursor-movement keys by changing the insertion point and handles the Delete key by simulating a Backspace key. The `OnKeyDown` method also calls `PositionCaret`.

The `PositionCaret` method is responsible for converting the insertion point (`iInsert`) into a pixel location for the caret. It does this using `MeasureString`. Unfortunately, the default version of `MeasureString` doesn't offer quite the precision required in applications like this. The most blatant
problem is that MeasureString normally excludes trailing blanks when calculating string lengths. To correct this problem, the program uses a version of MeasureString with a StringFormat argument and includes the enumeration value StringFormatFlags.MeasureTrailingSpaces in the FormatFlags property of StringFormat. Before that change, the caret would move whenever I typed letters that made up a word, but not when I typed a space after the word.

But that change isn't sufficient to make the caret line up nicely with displayed text. For reasons I discuss in Chapter 9, in the section "Grid Fitting and Text Fitting," the MeasureString and DrawString methods normally have built-in padding to compensate for problems related to the device-independent rasterization of outline fonts. To override this default behavior, the program uses a StringFormat object that it obtains from the static StringFormat.GenericTypographic property. As part of this solution (covered in Chapter 9), the program also uses the Graphics property TextRenderingHint to enable anti-aliasing of the text output.

**Right-to-Left Problems**

I mentioned earlier that the TypeAway program has almost the full functionality of a TextBox control in single-line mode. One problem is that it doesn't have clipboard support. Another is that TypeAway doesn't correctly display the caret when you type text that is written right to left, such as Arabic or Hebrew.

Let's take a look: run TypeAway, and switch to the Hebrew keyboard layout. We're going to type the Hebrew for "good morning," which is בֹּכֶר תוֹב, commonly transliterated as boker tov. To accomplish this little feat on an English keyboard, you first need to know how the characters of the Hebrew alphabet correspond to the keys of the keyboard.

**Hebrew Alphabet**

<table>
<thead>
<tr>
<th>Unicode</th>
<th>Letter</th>
<th>Glyph</th>
<th>Key</th>
<th>Unicode</th>
<th>Letter</th>
<th>Glyph</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05D0</td>
<td>alef</td>
<td>א</td>
<td>t</td>
<td>0x05DE</td>
<td>mem</td>
<td>מ</td>
<td>n</td>
</tr>
<tr>
<td>0x05D1</td>
<td>bet</td>
<td>ב</td>
<td>c</td>
<td>0x05DF</td>
<td>final nun</td>
<td>ל</td>
<td>i</td>
</tr>
<tr>
<td>0x05D2</td>
<td>gimel</td>
<td>ג</td>
<td>d</td>
<td>0x05E0</td>
<td>nun</td>
<td>נ</td>
<td>b</td>
</tr>
<tr>
<td>0x05D3</td>
<td>dalet</td>
<td>ד</td>
<td>s</td>
<td>0x05E1</td>
<td>samekh</td>
<td>ס</td>
<td>x</td>
</tr>
<tr>
<td>0x05D4</td>
<td>he</td>
<td>ה</td>
<td>v</td>
<td>0x05E2</td>
<td>ayin</td>
<td>י</td>
<td>g</td>
</tr>
<tr>
<td>0x05D5</td>
<td>vav</td>
<td>ו</td>
<td>u</td>
<td>0x05E3</td>
<td>final pe</td>
<td>נ</td>
<td>;</td>
</tr>
<tr>
<td>0x05D6</td>
<td>zayin</td>
<td>ז</td>
<td>z</td>
<td>0x05E4</td>
<td>pe</td>
<td>פ</td>
<td>p</td>
</tr>
<tr>
<td>0x05D7</td>
<td>het</td>
<td>ח</td>
<td>j</td>
<td>0x05E5</td>
<td>final tsadi</td>
<td>צ</td>
<td>.</td>
</tr>
<tr>
<td>0x05D8</td>
<td>tet</td>
<td>ט</td>
<td>y</td>
<td>0x05E6</td>
<td>tsadi</td>
<td>ש</td>
<td>m</td>
</tr>
<tr>
<td>0x05D9</td>
<td>yod</td>
<td>י</td>
<td>h</td>
<td>0x05E7</td>
<td>qof</td>
<td>ק</td>
<td>e</td>
</tr>
<tr>
<td>0x05DA</td>
<td>final kaf</td>
<td>כ</td>
<td>l</td>
<td>0x05E8</td>
<td>resh</td>
<td>ר</td>
<td>r</td>
</tr>
<tr>
<td>0x05DB</td>
<td>kaf</td>
<td>כ</td>
<td>f</td>
<td>0x05E9</td>
<td>shin</td>
<td>ש</td>
<td>a</td>
</tr>
<tr>
<td>0x05DC</td>
<td>lamed</td>
<td>ל</td>
<td>k</td>
<td>0x05EA</td>
<td>tav</td>
<td>ת</td>
<td>,</td>
</tr>
<tr>
<td>0x05DD</td>
<td>final mem</td>
<td>מ</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I've taken the spellings of these letters from The Unicode Standard Version 3.0. You'll note that some letter names include the word final. These letters are written differently when they fall at the end of a word.
You also need to know that Hebrew is written from right to left. So to type the Hebrew phrase shown above into TypeAway, you need to type the letters in this order: *bet* (the c key), *qof* (the e key), *resh* (the r key), space, *tet* (the y key), *vav* (the u key), and *bet* (the c key). TypeAway stores the Unicode characters in the character string in the order that you type them. That is correct. The *DrawString* method displays these characters from right to left. That is also correct, and the *DrawString* method must be given full credit and congratulations for recognizing and properly displaying text that reads right to left.

And now you know why TypeAway has to completely erase the line of previously drawn text: new text may not necessarily be appended at the end of the text string. When you're typing from the Hebrew keyboard, new text must be displayed at the beginning of the text string rather than the end. Typing in Arabic is even more critical: adjacent characters in Arabic are often joined to form different glyphs. *DrawString* needs to draw the whole string, not just individual characters, to correctly handle this situation.

Where TypeAway fails is in the caret positioning. When you're typing right-to-left text, the caret isn't showing the insertion point. The solution to this problem isn't trivial, particularly when you're dealing with a combination of left-to-right and right-to-left text in the same line. It appears that Windows Forms doesn't make available sufficient tools to solve this problem, but if you're interested in seeing how it's done using the Win32 API, check out the article "Supporting Multilanguage Text Layout and Complex Scripts with Windows NT 5.0" from the November 1998 issue of *Microsoft Systems Journal*. 

Chapter 7: Pages and Transforms

Overview

A primary goal in any graphics programming environment is device independence. Programs should be able to run without change on many different types of video display adapters regardless of the resolution. Programs should also be able to print text and graphics on many different printers without requiring a multitude of specialized printer drivers or completely separate drawing code.

In Chapter 5, I demonstrated that you can write graphics output code that draws to both the video display and the printer. Yet so far, I've been drawing in units of pixels—at least on the video display; the printer is something of a puzzle just yet—and pixels hardly seem device independent.

Device Independence Through Text

With some care, it's possible to use pixels in a device-independent manner. One way is to base graphics output on the default size of the Font property associated with the form. This approach is particularly useful if you're combining some simple graphics with text.

For example, suppose you were programming a simple database application using an index card metaphor. Each record is displayed as a simulated 3-by-5-inch index card. How large are the index cards in pixels? Think of a typewriter. A typewriter with a pica typeface types 6 lines to the inch, so an index card fits 18 lines of type. You can thus make the height of the index card equal to \(18 \times \text{Font.Height}\) pixels. You set the width to \(\frac{5}{3}\) times that number.

Making the width of the index card \(\frac{5}{3}\) times the height implies that the horizontal resolution of your output device—the number of pixels corresponding to a given measurement such as an inch—is equal to the vertical resolution. When a graphics output device has equal horizontal and vertical resolution, it is sometimes said to have square pixels. The very early video displays used when Windows was first released in 1985 did not have square pixels; it wasn't until the 1987 introduction of the IBM Video Graphics Array (VGA) that square pixels started to become a standard for PC-compatible video adapters.

Today, it's fairly safe to assume that the video display on which your Windows Forms program is running has square pixels. I say "fairly safe" because Windows doesn't require square pixels, and it's always possible that somebody may write a Windows device driver for some specialized display adapter that doesn't have square pixels.

Printers these days often do not have square pixels. Often the resolution in one dimension is twice the resolution in the other.

How Much Is That in Real Money?

Let's start exploring the relationship between pixels and real-world measurements. Suppose you draw a box of 100 pixels width and height located 100 pixels from the upper left corner of the client area (or printable area of the printer page).

HundredPixelsSquare.cs

```csharp
//--------------------------------------------------
// HundredPixelsSquare.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HundredPixelsSquare: PrintableForm
{
    public new static void Main()
    {
```
Application.Run(new HundredPixelsSquare());
}

public HundredPixelsSquare()
{
    Text = "Hundred Pixels Square";
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.FillRectangle(new SolidBrush(clr), 100, 100, 100, 100);
}

How large is that box on the screen? How large is it on the printer? Is it even square?

Certainly you have a vague idea of how large this box will be on the screen without actually running the program, at least in terms of the relationship of the size of the box to the size of the screen. The smallest video display size in common use today is 640 pixels horizontally by 480 pixels vertically (or 640 × 480). On such a display, the box would be roughly 1/6 the width of the screen and 1/5 the height. But video displays these days can go up to 2048 × 1536 pixels or so, in which case the box is much smaller in relationship to the entire screen.

It would be nice to know the resolution of the video display, perhaps in a common measurement like dots per inch (dpi). However, while such a concept is very well defined for printers—it’s usually printed right on the box you take home from the computer store—it remains an elusive concept for video displays. If you think about it, the actual dpi resolution of a video display is based on two measurements: the physical size of the monitor (usually measured diagonally in inches) and the corresponding pixel dimensions.

Confusingly enough, this latter measurement is often termed the resolution of the video display. In the SystemInformation class, it’s the item called PrimaryMonitorSize. If you invoke the Display Properties dialog box—which you can run from Control Panel or by right-clicking on your desktop and selecting Properties from the menu—and select the Settings tab, this pixel dimension is called the screen area, and that’s the term I’ll use.

Video display adapters these days are capable of half a dozen (or so) different screen-area settings, and video monitors come in several different sizes. Here’s a little table that shows the approximate video resolution in dots per inch for various combinations of monitor sizes and screen areas:

<table>
<thead>
<tr>
<th>Actual Video Resolution (dots per inch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor Size (diagonally)</td>
</tr>
<tr>
<td>Screen Area</td>
</tr>
<tr>
<td>640 × 480</td>
</tr>
<tr>
<td>800 × 600</td>
</tr>
<tr>
<td>1024 × 768</td>
</tr>
<tr>
<td>1152 × 870</td>
</tr>
<tr>
<td>1280 × 1024</td>
</tr>
<tr>
<td>1600 × 1200</td>
</tr>
<tr>
<td>2048 × 1536</td>
</tr>
</tbody>
</table>

I’m assuming that the actual display area is an inch smaller than the rated diagonal size and that the monitor has the standard aspect ratio of 4:3. For example, a 21-inch monitor has a diagonal display area of 20 inches, implying (thank you, Pythagoras) dimensions of 16 inches horizontally and 12 inches vertically. For screen areas of 1152 × 870 and 1280 × 1024, the horizontal and vertical
dimensions are not in the ratio of 4:3 and hence the horizontal and vertical resolutions are unequal—but they're close enough to assume they're equal.

So if you were running a 1600 × 1200 video mode on a 21-inch monitor, that 100-pixel square box would be about 1-inch square. But it could be almost as small as 1/2 inch or larger than 2 inches. Of course, few people use 21-inch monitors to run a 640 × 480 video mode, nor do they try to run 2048 × 1536 on 15-inch monitors. The more likely range of resolutions appears in the diagonal area of the table from the upper left to the lower right.

Windows doesn't know the size of your monitor, so it can't tell you the actual resolution of your video display. And even if it did know your monitor's size, what would it do when you connect a video projector to your machine and create a screen some 6 feet wide? What should it do? Should Windows assume a much lower resolution because the screen is larger? Almost assuredly, you don't want that.

The most essential issue regarding the video display is readable text. The default font should be large enough for you to read, obviously, but it shouldn't be much larger because you also want to fit as much text on the screen as possible.

For this reason, Windows basically ignores screen area and monitor size and instead delegates the selection of a video resolution to a very important person: You!

I've already mentioned the Display Properties dialog box. The Settings tab lets you change your video display settings. (Note that this description of Display Properties is based on Windows 2000. Other versions of Windows might differ slightly.) The Settings page also has a button labeled Advanced. In a roundabout way, this button essentially lets you select a video resolution in dots per inch for the video display. I say this is "roundabout" because you actually select a Windows system font size that is comfortable for you to read. That system font is assumed to be 10 points in size. (Fonts are measured in points, which in computer typography are 1/72 inch.) The pixel size of the 10-point font that you select implies a resolution of the video display in dots per inch.

For example, the default system font is called Small Fonts. The Small Fonts characters are 13 pixels in height. If that font is assumed to be 10 points in size, then 13 pixels are equivalent to 10/72 inch, and the display resolution (with a little rounding involved) is 96 dpi.

One common alternative to Small Fonts is Large Fonts, which are 16 pixels tall. If these 16 pixels are equivalent to 10/72 inch, then the display resolution (again with a little rounding) is 120 dpi.

By the way, the Windows system font is *not* the default font that's accessible through the Font property in a Windows Forms program. Windows Forms sets the default font to something a bit smaller—about 8 points in size.¹

Large Fonts and Small Fonts are not the only choices. By selecting Other from the list box, you are presented with a ruler that you can manually adjust to pick a really big font (implying a resolution of 480 dpi) or a really small font (about 19 dpi).

Here's a quick summary: You select a system font size that you find readable. Windows assumes that this is a 10-point font. That implies a video display resolution in dots per inch.

Commonly, the system font you choose will have larger physical dimensions than the point size implies. When people read print on paper, the distance between the eyes and the text is generally about a foot, but a video display is often viewed from a distance of 2 feet or so.

¹ Veteran Windows programmers might be curious about where my numbers come from. I'm quoting the TextMetric field tmHeight (which is 16 for Small Fonts and 20 for Large Fonts) less tmInternalLeading (which is 3 and 4, respectively). The tmHeight value is suitable for line spacing; tmHeight less tmInternalLeading indicates the point size converted to pixels (13 for Small Fonts and 16 for Large Fonts). Somewhat confusingly, the default font in Windows Forms has a Font.Height property that reports similar values: 13 for Small Fonts and 15 for Large Fonts. But this is a line-spacing value that is comparable with tmHeight. The Windows system font is 10 points; the default Windows Forms font is about 8 points.

**Dots Per Inch**

The Graphics object has two properties that indicate the resolution of the graphics output device in dots per inch:
Here's a short program that displays these values without much fuss.

DotsPerInch.cs

    //------------------------------------------
    // DotsPerInch.cs © 2001 by Charles Petzold
    //------------------------------------------
    using System;
    using System.Drawing;
    using System.Windows.Forms;

    class DotsPerInch: PrintableForm
    {
        public new static void Main()
        {
            Application.Run(new DotsPerInch());
        }
        public DotsPerInch()
        {
            Text = "Dots Per Inch";
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            grfx.DrawString(String.Format("DpiX = {0}\nDpiY = {1}",
                                           grfx.DpiX, grfx.DpiY),
                           Font, new SolidBrush(clr), 0, 0);
        }
    }

The values that this program reports in its client area are the same as the settings you've made in the Display Properties dialog box: 96 dpi if you've selected Small Fonts, 120 dpi if you've selected Large Fonts, and something else if you've selected a custom size.

If you click on the client area, the printed version will show the resolution of your printer, which is probably something you already knew or could look up in the manual. Printers these days have resolutions of 300, 600, 1200, or 2400 dpi, or 720, 1440, or 2880 dpi.

**What's with the Printer?**

Earlier in this chapter, I presented the HundredPixelsSquare program that displayed a box 100 pixels square. I wondered how large the box was on the screen. The real answer is that the physical dimensions of this box are irrelevant. Nobody expects a ruler held up to the screen to provide much useful information. The important point is that rulers displayed on the screen should be consistent with each other. In that sense, the horizontal and vertical screen dimensions of a 100-pixel square box in inches are
That is, 1.04 inches if you've selected Small Fonts, 0.83 inch if you've selected Large Fonts, and something else if you've selected a custom size.

And on the printer... Well, you may want to try this one yourself. On your printer, the HundredPixelsSquare program draws a box that is precisely 1 inch square. Let's try something else. This program attempts to draw an ellipse with a diameter of 1 inch based on the DpiX and DpiY properties of the Graphics object.

TryOneInchEllipse.cs
//------------------------------------------------
// TryOneInchEllipse.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TryOneInchEllipse: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TryOneInchEllipse());
    }
    public TryOneInchEllipse()
    {
        Text = "Try One-Inch Ellipse";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawEllipse(new Pen(clr), 0, 0, grfx.DpiX, grfx.DpiY);
    }
}

On the video display, the size of this ellipse looks about right. On my 600 dpi printer, however, the ellipse is 6 inches in diameter.

For the video display, the coordinates you pass to the Graphics drawing functions are obviously in units of pixels. For the printer, however, that is apparently not the case. For the printer, the coordinates passed to the Graphics drawing functions are actually interpreted as units of 0.01 inch regardless of the printer. We'll see how this works shortly. The nice part is that the resolution of the video display is probably somewhere in the region of 100 dpi, and the printer is treated as if it were a 100-dpi device. This means that in a pinch you can use the same coordinates when displaying graphics on the video display and the printer, and you'll get approximately the same results.

Manual Conversions

If you wanted to, you could use the DpiX and DpiY properties of the Graphics object to adjust the coordinates that you pass to the drawing functions. For example, suppose you wanted to use floating-point coordinates to draw in units of millimeters. You'd need a method that converts from millimeters to pixels:

PointF MMConv(Graphics grfx, PointF pointf)
{
pointf.X *= grfx.DpiX / 25.4f;
pointf.Y *= grfx.DpiY / 25.4f;

return pointf;
}

The point you're passing to this method is your desired units of millimeters. Dividing that point by 25.4 converts it to inches. (That's an exact calculation, by the way.) Multiplying by the resolution in dots per inch converts it to pixels.

Just so we're sure about this, let's draw a 10-centimeter ruler.  

// TenCentimeterRuler.cs
// TenCentimeterRuler.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class TenCentimeterRuler: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TenCentimeterRuler());
    }
    public TenCentimeterRuler()
    {
        Text = "Ten-Centimeter Ruler";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr);
        Brush brush = new SolidBrush(clr);
        const int xOffset = 10;
        const int yOffset = 10;

        grfx.DrawPolygon(pen, new PointF[]
        {
            MMConv(grfx, new PointF(xOffset, yOffset)),
            MMConv(grfx, new PointF(xOffset + 100, yOffset)),
            MMConv(grfx, new PointF(xOffset + 100, yOffset + 10)),
            MMConv(grfx, new PointF(xOffset, yOffset + 10))
        });

        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = StringAlignment.Center;
for (int i = 1; i < 100; i++)
{
    if (i % 10 == 0)  // Centimeter markings
    {
        grfx.DrawLine(pen,
                      MMConv(grfx, new PointF(xOffset + i, yOffset)),
                      MMConv(grfx, new PointF(xOffset + i, yOffset + 5)));

        grfx.DrawString((i/10).ToString(), Font, brush,
                         MMConv(grfx, new PointF(xOffset + i, yOffset + 5)),
                         strfmt);
    }
    else if (i % 5 == 0)  // Half-centimeter markings
    {
        grfx.DrawLine(pen,
                      MMConv(grfx, new PointF(xOffset + i, yOffset)),
                      MMConv(grfx, new PointF(xOffset + i, yOffset + 3)));
    }
    else  // Millimeter markings
    {
        grfx.DrawLine(pen,
                      MMConv(grfx, new PointF(xOffset + i, yOffset)),
                      MMConv(grfx, new PointF(xOffset + i, yOffset + 2.5f)));
    }
}

PointF MMConv(Graphics grfx, PointF pointf)
{
    pointf.X *= grfx.DpiX / 25.4f;
    pointf.Y *= grfx.DpiY / 25.4f;

    return pointf;
}

Here's how the ruler looks on the screen:
This diagram also involves some text. How did I know the text was going to look right? Well, I know that the Font property is about an 8-point font, so I know that the font characters should be about 3 millimeters tall, which is about the right size.

I've made the TenCentimeterRuler class a descendent of PrintableForm to hammer home a point: this technique will not work on the printer. My 600-dpi printer displays it six times too large.

**Page Units and Page Scale**

So that you can avoid writing methods such as MMConv, GDI+ includes a facility that performs automatic scaling to dimensions of your choosing. Basically, the coordinates you pass to the Graphics drawing functions are scaled by constants, just as in the MMConv method. But you don't set these scaling factors directly. Instead, you set them indirectly using two properties of the Graphics class named PageUnit and PageScale:

```
<table>
<thead>
<tr>
<th>Graphics Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>GraphicsUnit</td>
</tr>
<tr>
<td>float</td>
</tr>
</tbody>
</table>
```

You set the PageUnit property to a value in the GraphicsUnit enumeration:

```
<table>
<thead>
<tr>
<th>GraphicsUnit Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>World</td>
</tr>
<tr>
<td>Display</td>
</tr>
<tr>
<td>Pixel</td>
</tr>
<tr>
<td>Point</td>
</tr>
<tr>
<td>Inch</td>
</tr>
<tr>
<td>Document</td>
</tr>
<tr>
<td>Millimeter</td>
</tr>
</tbody>
</table>
```

If you say, for example, "I want to draw in units of hundredths of an inch," you then set these two properties like so:

```csharp
grfx.PageUnit = GraphicsUnit.Inch;
grfx.PageScale = 0.01f;
```

This is equivalent to saying, "When I specify a coordinate of 1, I want it to equal 0.01 inch." Following these calls, this DrawLine method draws a 1-inch-long line:

```csharp
grfx.DrawLine(pen, 0, 0, 100, 0);
```

That's an actual measurable inch on the printer and equal to grfx.DpiX pixels on the video display. You'll get the same results with

```csharp
grfx.PageUnit = GraphicsUnit.Document;
grfx.PageScale = 3;
```

or

```csharp
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 0.254f;
```

or

```csharp
grfx.PageUnit = GraphicsUnit.Point;
grfx.PageScale = 0.72f;
```
The default settings are `GraphicsUnit.Pixel` for the video display and `GraphicsUnit.Display` for the printer, both with a `PageSize` of 1. Notice that the `GraphicsUnit.Display` value means something different for the video display than for the printer. For the video display, it's the same as `GraphicsUnit.Pixel`, but for the printer, `GraphicsUnit.Display` indicates units of 1/100 inch.

So if we want to get that `TenCentimeterRuler` program to work on the printer, we need to set `PageUnit` to `GraphicsUnit.Pixel` and everything should be OK. Let's do that by defining a class that inherits from `TenCentimeterRuler`. The new `OnPage` method here resets the `PageUnit` property and then calls the base `DoPage` class.

```csharp
public new static void Main()
{
    Application.Run(new PrintableTenCentimeterRuler());
}

public PrintableTenCentimeterRuler()
{
    Text = "Printable " + Text;
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.PageUnit = GraphicsUnit.Pixel;
    base.DoPage(grfx, clr, cx, cy);
}
```

This program doesn't use the `cx` and `cy` arguments to `DoPage`. These dimensions—of the form's client area and of the printable area of the printer page—are in units compatible with the default `PageUnit`. In the general case, if you change `PageUnit`, you'll probably need to recalculate the size of the output device in identical units. I'll discuss this problem shortly.

Even though we're now drawing to the printer in units of pixels, the font still looks good. The font accessible from the `Font` property of the form is an 8-point font on the video display, and it's an 8-point font on the printer as well. We'll see how this works in Chapter 9.

This program still has a problem, however, one involving the pen that the `TenCentimeterRuler` version of `DoPage` defines:

```csharp
Pen pen = new Pen(clr);
```

This pen gets a default width of 1. On the video display, that means a width of 1 pixel. On the printer, that's normally a width of 1/100 inch. However, if you change `PageUnit` to `GraphicsUnit.Pixel`, the 1-unit-wide pen is now interpreted as a width of 1 pixel. On some very high-resolution printers, the ruler may be nearly invisible.
Rather than continuing to mess around with the original 10-centimeter-ruler program, let's take advantage of the `PageUnit` and `PageScale` properties to eliminate the manual conversion.

**TenCentimeterRulerAuto.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class TenCentimeterRulerAuto : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TenCentimeterRulerAuto());
    }

    public TenCentimeterRulerAuto()
    {
        Text = "Ten-Centimeter Ruler (Auto)";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr, 0.25f);
        Brush brush = new SolidBrush(clr);
        const int xOffset = 10;
        const int yOffset = 10;

        grfx.PageUnit = GraphicsUnit.Millimeter;
        grfx.PageScale = 1;
        grfx.DrawRectangle(pen, xOffset, yOffset, 100, 10);

        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = StringAlignment.Center;
        for (int i = 1; i < 100; i++)
        {
            if (i % 10 == 0) // Centimeter markings
            {
                grfx.DrawLine(pen,
                    new PointF(xOffset + i, yOffset),
                    new PointF(xOffset + i, yOffset + 5));

                grfx.DrawString((i/10).ToString(), Font, brush,
                    new PointF(xOffset + i, yOffset + 5),
                    strfmt);
            }
        }
    }
}
```

else if (i % 5 == 0)  // Half-centimeter markings
{
    grfx.DrawLine(pen,
                  new PointF(xOffset + i, yOffset),
                  new PointF(xOffset + i, yOffset + 3));
}
else                     // Millimeter markings
{
    grfx.DrawLine(pen,
                  new PointF(xOffset + i, yOffset),
                  new PointF(xOffset + i, yOffset + 2.5f));
}
}

Besides eliminating the MMConv method, I've really made just a few changes. My MMConv method worked only with PointF structures, so in the earlier ruler-drawing programs, I used DrawPolygon rather than DrawRectangle to draw the outline of the ruler. Because GDI+ scales both coordinates and sizes in the same way, I can use DrawRectangle here. Another change occurs toward the beginning of the DoPage method, where the program creates a pen 0.25 unit wide:
Pen pen = new Pen(clr, 0.25f);
The program also sets up the Graphics object to draw in units of millimeters:
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
You might wonder if it makes a difference whether you set the PageUnit and PageScale properties before you create the pen or if you create a pen with a specific width before you set the PageUnit and PageScale properties. It doesn't matter. Pens are device independent! They are not associated with a particular Graphics object until the call to one of the line-drawing methods. Only at that time is the pen width interpreted in units indicated by the current PageUnit and PageScale properties. In this case, the pen is interpreted to be 0.25 millimeter or about 1/100 inch. You may want to try a smaller value (such as 0.10 millimeter) to see the difference on the printer.

If you don't include a width in the pen constructor, the pen is created 1 unit wide, which in this case means that the pen is 1 whole millimeter wide and the ruler divisions become one big blob. (Try it!)

Pen Widths

What's a proper pen width for the printer? You might take a cue from PostScript—the well-known and highly respected page composition language many upscale printers use—and think of a normal default pen width as 1 point, otherwise expressible as 1/72 inch, or about 1/3 millimeter. I personally find a 1-point pen width to be a bit on the chunky side, but it's an easy rule to remember.

Here's a program that displays a bunch of pen widths in units of points.

PenWidths.cs
//----------------------------------------
// PenWidths.cs © 2001 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PenWidths: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new PenWidths());
    }
    public PenWidths()
    {
        Text = "Pen Widths";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        float y = 0;

        grfx.PageUnit = GraphicsUnit.Point;
        grfx.PageScale = 1;

        for (float f = 0; f < 3.2; f += 0.2f)
        {
            Pen pen = new Pen(clr, f);
            string str = String.Format("{0:F1} point wide pen: ", f);
            SizeF sizef = grfx.MeasureString(str, Font);

            grfx.DrawString(str, Font, brush, 0, y);
            grfx.DrawLine(pen, sizef.Width, y + sizef.Height / 2, sizef.Width + 144, y + sizef.Height / 2);
            y += sizef.Height;
        }
    }
}

Of course, although you can specify pen widths that are fractions of pixels, they can be rendered only with whole pixel widths. On the video display, many of the pen widths created by this program round to the same values:
One thing you don’t have to worry about on the video display is the pen width rounding down to 0 and the pen disappearing. Pens are always drawn at least 1 pixel wide. Indeed, you can set the width to 0 in the \textit{Pen} constructor and always get 1-pixel-wide lines regardless of the \textit{PageUnit} and \textit{PageScale} properties.

Although 0-width pens are fine for the video display, they should never be used on the printer. On today's high-resolution laser printers, 1-pixel-wide lines are virtually invisible.

Here's a program for a ruler marked in inches that uses units of 1/64 inch and creates a pen 1/128 inch wide.

\begin{verbatim}
using System;
using System.Drawing;
using System.Windows.Forms;

class SixInchRuler: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new SixInchRuler());
    }
    public SixInchRuler()
    {
        Text = "Six-Inch Ruler";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        Pen pen = new Pen(clr, 0.5f);
        const int xOffset = 16;
        const int yOffset = 16;
        grfx.PageUnit = GraphicsUnit.Inch;
        grfx.PageScale = 1 / 64f;
        grfx.DrawRectangle(pen, xOffset, yOffset, 6 * 64, 64);
    }
}
\end{verbatim}
StringFormat strfmt = new StringFormat();
strfmt.Alignment = StringAlignment.Center;

for (int i = 1; i < 6 * 16; i++)
{
    int x = xOffset + i * 4;
    int y = yOffset;
    int dy;

    if(i % 16 == 0)          // One-inch marks
    {
        dy = 32;
        grfx.DrawString((i / 16).ToString(), Font, brush,
                        x, y + dy, strfmt);
    }
    else if(i % 8 == 0)      // Half-inch marks
        dy = 24;
    else if(i % 4 == 0)      // Quarter-inch marks
        dy = 20;
    else if(i % 2 == 0)      // Eighth-inch marks
        dy = 16;
    else                     // Sixteenth-inch marks
        dy = 12;

    grfx.DrawLine(pen, x, y, x, y + dy);
}
}

The ruler looks like this:

![Image of a ruler]

You might have noticed that text seems unaffected by any PageUnit and PageScale values I've set. That's because the font accessible through the form's Font property is about 8 points in size and remains that same size. In Chapter 9, I'll show you how to create a font that is affected by the PageUnit and PageScale properties.

Page Transforms

What you're effectively setting when you set the PageScale and PageUnit properties is known as the page transform. This transform can be represented by a pair of simple formulas. Assume that the
coordinates you pass to the Graphics drawing methods are page coordinates. (This assumption isn't quite true, as you'll see later in this chapter, but it is true if you're setting only the PageScale and PageUnit properties.) You can represent a point in page units as \((x_{\text{page}}, y_{\text{page}})\).

Pixel coordinates relative to the upper left corner of the client area (or the upper left corner of the printable area of the page) are said to be in device coordinates, or \((x_{\text{device}}, y_{\text{device}})\). The page transform depends on the PageUnit, PageScale, DpiX, and DpiY properties.

### Page Transform Formulas

<table>
<thead>
<tr>
<th>PageUnit Value</th>
<th>Translation Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsUnit.Pixel</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} )</td>
</tr>
<tr>
<td>GraphicsUnit.Display (video display)</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} )</td>
</tr>
<tr>
<td>GraphicsUnit.Display (printer)</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} \times \text{DpiX} / 100 )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} \times \text{DpiY} / 100 )</td>
</tr>
<tr>
<td>GraphicsUnit.Inch</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} \times \text{DpiX} )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} \times \text{DpiY} )</td>
</tr>
<tr>
<td>GraphicsUnit.Millimeter</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} \times \text{DpiX} / 25.4 )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} \times \text{DpiY} / 25.4 )</td>
</tr>
<tr>
<td>GraphicsUnit.Point</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} \times \text{DpiX} / 72 )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} \times \text{DpiY} / 72 )</td>
</tr>
<tr>
<td>GraphicsUnit.Document</td>
<td>(x_{\text{device}} = x_{\text{page}} \times \text{PageScale} \times \text{DpiX} / 300 )</td>
</tr>
<tr>
<td></td>
<td>(y_{\text{device}} = y_{\text{page}} \times \text{PageScale} \times \text{DpiY} / 300 )</td>
</tr>
</tbody>
</table>

In general,

\[
\begin{align*}
  x_{\text{device}} &= x_{\text{page}} \times \text{PageScale} \times \text{DpiX} / (\text{GraphicsUnit units per inch}) \\
  y_{\text{device}} &= y_{\text{page}} \times \text{PageScale} \times \text{DpiY} / (\text{GraphicsUnit units per inch})
\end{align*}
\]

Watch out for integer arithmetic. You might try to set the page scale to 1/64 of an inch like so:

```csharp
grfx.PageScale = 1 / 64;
```

However, C# will perform the integer division before automatically converting to float! Help out the compiler by telling it what you want to happen:

```csharp
grfx.PageScale = 1f / 64;
```

The page transform affects all the coordinates of all the drawing functions implemented in the Graphics class that I've discussed so far. It also affects the information returned from MeasureString and the version of the GetHeight method implemented in the Font class that takes a Graphics object argument.

The page transform is a characteristic of the Graphics class. The page transform doesn't affect anything that's not a member of the Graphics class or that (unlike GetHeight) doesn't have a Graphics object argument. The page transform doesn't affect the information you get from ClientSize, for example. ClientSize is always in units of pixels.

### Saving the Graphics State

Setting the PageUnit and PageScale properties of the Graphics object profoundly affects the subsequent display of graphics. You might want to set these properties—or other properties in the Graphics class—to draw some graphics or obtain some information, and then revert back to the original properties.

The Graphics class has two methods, named Save and Restore, that let you do just that: save the properties of the Graphics object and later restore them. These two methods use the GraphicsState class from the namespace System.Drawing.Drawing2D.
The *GraphicsState* class has nothing public of any interest. You really treat it as a black box. When you call

```csharp
GraphicsState gs = grfx.Save();
```

all the current read/write properties of the *Graphics* object are stored in the *GraphicsState* object. You can then change those properties on the *Graphics* object. To restore the saved properties, use

```csharp
grfx.Restore(gs);
```

Programmers with experience using Win32 are probably accustomed to thinking of the similar facility (involving the functions *SaveDC* and *RestoreDC*) in terms of a last-in-first-out stack. The Windows Forms implementation is more flexible. For example, you could begin *OnPaint* processing by defining three different graphics states:

```csharp
GraphicsState gs1 = grfx.Save();

    // Change some properties.
    :

GraphicsState gs2 = grfx.Save();

    // Change some properties.
    :

GraphicsState gs3 = grfx.Save();
```

You could then arbitrarily and in any order make calls to the *Restore* method to use any one of these three graphics states.

A similar facility is provided by the *BeginContainer* and *EndContainer* methods of the *Graphics* class. These methods make use of the *GraphicsContainer* class in *System.Drawing.Drawing2D*.

### Metrical Dimensions

The dimensions of a form’s client area are available from the *ClientSize* property. These dimensions are always in units of pixels. If you set a new page transform, you probably want the dimensions of the client area not in units of pixels but in units corresponding to what you’re now using in the drawing methods.

There are at least two ways to get the client size in metrical dimensions. Probably the most convenient way is the *VisibleClipBounds* property of the *Graphics* object. This property always returns the dimensions of the client area in units consistent with the current settings of the *PageUnit* and *PageScale* properties. Here’s a program that uses this information to show the size of the client area using all the possible units.

### WhatSize.cs

```csharp
// WhatSize.cs © 2001 by Charles Petzold
// ------------------------------
using System;
using System.Drawing;
```
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WhatSize: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new WhatSize());
    }

    public WhatSize()
    {
        Text = "What Size?";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        int y = 0;

        DoIt(grfx, brush, ref y, GraphicsUnit.Pixel);
        DoIt(grfx, brush, ref y, GraphicsUnit.Display);
        DoIt(grfx, brush, ref y, GraphicsUnit.Document);
        DoIt(grfx, brush, ref y, GraphicsUnit.Inch);
        DoIt(grfx, brush, ref y, GraphicsUnit.Millimeter);
        DoIt(grfx, brush, ref y, GraphicsUnit.Point);
    }

    void DoIt(Graphics grfx, Brush brush, ref int y, GraphicsUnit gu)
    {
        GraphicsState gs = grfx.Save();

        grfx.PageUnit = gu;
        grfx.PageScale = 1;

        SizeF sizef = grfx.VisibleClipBounds.Size;

        grfx.Restore(gs);

        grfx.DrawString(gu + " : " + sizef, Font, brush, 0, y);
        y += (int) Math.Ceiling(Font.GetHeight(grfx));
    }
}

The DoIt method in WhatSize makes use of the Save and Restore facility so that the different PageUnit settings don't interfere with the actual display of information when we call the DrawString method and the GetHeight call. Here's a typical WhatSize display:
Unfortunately, the printer is different. For the printer, `VisibleClipBounds` is designed to return values in units of 1/100 inch regardless of the page transform. However, if the printer `PageUnit` is set for pixels, `VisibleClipBounds` returns the printable area of the page in pixels.

An historical note: I wrote the first how-to-program-for-Windows magazine article for the December 1986 issue of *Microsoft Systems Journal*. The sample program in that article was called WSZ ("what size"), and it displayed the size of the program's client area in pixels, inches, and millimeters. WhatSize is a somewhat simplified—and considerably shorter—version of that program.

Another approach to determining the size of the display area involves using the `TransformPoints` method implemented in the `Graphics` class:

```
Graphics TransformPoints Methods

void TransformPoints(CoordinateSpace csDst, CoordinateSpace csSrc, 
                     Point[] apt)
void TransformPoints(CoordinateSpace csDst, CoordinateSpace csSrc, 
                     PointF[] aptf)
```

The `CoordinateSpace` enumeration is defined in the `System.Drawing.Drawing2D` namespace:

```
CoordinateSpace Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>0</td>
</tr>
<tr>
<td>Page</td>
<td>1</td>
</tr>
<tr>
<td>Device</td>
<td>2</td>
</tr>
</tbody>
</table>
```

So far, we know of the coordinate space called `Device` (that's units of pixels relative to the upper left corner of the client area) and the coordinate space called `Page` (units of inches, millimeters, points, or such). If you have an array of `Point` or `PointF` structures in device units, you can convert those values to page units by calling

```
grfx.TransformPoints(CoordinateSpace.Page, CoordinateSpace.Device, apt);
```

I'll talk about the coordinate space known as `World` shortly.

Here's another version of the WhatSize program that uses `TransformPoints` to calculate the size of the client area.

```
WhatSizeTransform.cs
//------------------------------------------------
// WhatSizeTransform.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
```
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WhatSizeTransform : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new WhatSizeTransform());
    }
    public WhatSizeTransform()
    {
        Text = "What Size? With TransformPoints";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        int y = 0;
        Point[] apt = { new Point(cx, cy) };

        grfx.TransformPoints(CoordinateSpace.Device,
                              CoordinateSpace.Page, apt);

        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Pixel);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Display);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Document);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Inch);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Millimeter);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Point);
    }
    void DoIt(Graphics grfx, Brush brush, ref int y,
              Point pt, GraphicsUnit gu)
    {
        GraphicsState gs = grfx.Save();

        grfx.PageUnit = gu;
        grfx.PageScale = 1;

        PointF[] aptf = { pt };

        grfx.TransformPoints(CoordinateSpace.Page,
                              CoordinateSpace.Device, aptf);

        SizeF sizef = new SizeF(aptf[0]);
        grfx.Restore(gs);
    }
}
I've added an extra argument to the program's `DoIt` method: a `Point` structure containing the width and height of the display area in pixels. For the video display, that's not much of a problem because the `cx` and `cy` arguments to `DoPage` are already pixels. For the printer, however, they are not. For that reason, the `DoPage` method adds `cx` and `cy` to a `Point` structure, makes a single-element `Point` array, and passes that array to `TransformPoints` to convert the values to device units. Notice that for this call to `TransformPoints` the destination coordinate space is `CoordinateSpace.Device`. `DoIt` then uses `TransformPoints` to convert from device units to `CoordinateSpace.Page`.

**Arbitrary Coordinates**

Some of the graphics programs shown so far in this book have scaled their output to the size of the client area or the printable area of the printer page. Programs in this chapter have drawn in specific sizes in units of millimeters or inches.

Then there are times when you want to hard-code a bunch of coordinates and would prefer that you could skip any explicit scaling of them. For example, you may want to code some graphics output using a coordinate system of (say) 1000 units horizontally and 1000 units vertically. You want this coordinate system to be as large as possible but still fit inside your client area or the printer page.

This program demonstrates how to do just that.

**ArbitraryCoordinates.cs**

```csharp
// ArbitraryCoordinates.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;
class ArbitraryCoordinates: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new ArbitraryCoordinates());
    }
    public ArbitraryCoordinates()
    {
        Text = "Arbitrary Coordinates";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.PageUnit = GraphicsUnit.Pixel;
        SizeF sizef = grfx.VisibleClipBounds.Size;

        grfx.PageUnit  = GraphicsUnit.Inch;
        grfx.PageScale = Math.Min(sizef.Width  / grfx.DpiX / 1000,
                                  sizef.Height / grfx.DpiY / 1000);
```
grfx.DrawLine(new Pen(clr), 0, 0, 990, 990);
}
}

The `DoPage` method first sets `PageUnit` to `GraphicsUnit.Pixel` for the sole purpose of obtaining the `VisibleClipBounds` property indicating the size of the client area or printer page in units of pixels.

Next, `DoPage` sets `PageUnit` to inches:

```
grfx.PageUnit = GraphicsUnit.Inch;
```

Earlier I showed the following transform formulas that apply to a `PageUnit` of inches:

```
x_{device} = x_{page} \times \text{PageScale} \times \text{DpiX}
y_{device} = y_{page} \times \text{PageScale} \times \text{DpiY}
```

You want $x_{page}$ and $y_{page}$ to range from 0 through 1000 while $x_{device}$ and $y_{device}$ range from 0 through the `Width` and `Height` properties (respectively) from `VisibleClipBounds`. In other words,

```
\text{Width} = 1000 \times \text{PageScale} \times \text{DpiX}
\text{Height} = 1000 \times \text{PageScale} \times \text{DpiY}
```

However, these two equations would result in two different `PageScale` factors, and you can have only one. You want the lesser of the two calculated values:

```
grfx.PageScale = Math.Min(sizef.Width / grfx.DpiX / 1000,
                        sizef.Height / grfx.DpiY / 1000);
```

The program then draws an ellipse with a width and height of 990 units. (Using 1000 or 999 for the width and height sometimes causes one side of the figure to be truncated for large window sizes.) The resultant figure is a circle that appears at the left of the client area when the client area is wide and at the top when the client area is tall:

![Arbitrary Coordinates](image1.png)

You can also print the circle; it will be as large as the printable width of the page.

There's a subtle problem in this program, however. Try reducing the window size as far as it will go. You'll notice that there's a limit in the width of the window, but you can decrease the height of the window until the client area height is 0. At that point, you'll get an exception because the `DoPage` method will be setting the `PageScale` to 0, an invalid value.

You can deal with this problem in a couple ways. Perhaps the most obvious is simply to abort the `DoPage` method if the height of the client area is 0:

```
if (cy == 0)
    return;
```

That's not a problem because it doesn't make sense to draw anything anyway.
Don't you find it a little peculiar that you're getting a call to the `OnPaint` method anyway given that your client area is of 0 dimension? It wouldn't hurt to put a statement like this at the beginning of your `OnPaint` method:

```csharp
if (pea.ClipRectangle.IsEmpty)
    return;
```

This statement is equivalent:

```csharp
if (grfx.IsVisibleClipEmpty)
    return;
```

A very specialized solution is to use the `Math.Max` method in the calculation of the `PageScale` property to prevent values of 0:

```csharp
grfx.PageScale = Math.Min(sizef.Width / grfx.DpiX / 1000,
                           Math.Max(sizef.Height, 1) / grfx.DpiY / 1000);
```

Or, to demonstrate that you known something about exception handling in C#, you can put the statement in a `try` block:

```csharp
try
{
    grfx.PageScale = Math.Min(sizef.Width / grfx.DpiX / 1000,
                               sizef.Height / grfx.DpiY / 1000);
}
catch
{
    return;
}
```

But a method that might not seem so obvious is to prevent the client area from shrinking down to a 0 height in the first place. The static property `SystemInformation.MinimumWindowSize` returns a size whose height is simply the sum of the caption bar height and twice the sizing border height. The width is considerably greater to give windows a minimum width that still allows part of the program's caption bar to be visible.

You can set a form's `MinimumSize` property to keep the window above a certain dimension. Try putting this in the constructor for `ArbitraryCoordinates`:

```csharp
MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);
```

### What You Can't Do

There are several things you can't do with the page transform. First, you can't set `PageScale` to negative values; that is, you can't make x coordinates increase to the left (which few people want to do anyway) or y coordinates increase going up the screen (which is something that's useful for the mathematically inclined).

Second, you can't have different units in the horizontal and vertical directions. The `PageScale` and `PageUnit` properties apply to both axes equally. A function like

```csharp
grfx.DrawEllipse(pen, 0, 0, 100, 100);
```

will always draw a circle regardless of the page transform, with one exception: when you set a `PageUnit` of `GraphicsUnit.Pixel` and your output device has different horizontal and vertical resolution. This issue will rarely come up for the video display, but it's fairly common for printers.

And finally, you can't change the origin. The point (0, 0) in page coordinates always maps to the upper left corner of the client area or printable area of the printer page.

Fortunately, there's another transform supported by GDI+ that lets you do all of these tasks and more.
Hello, World Transform

The other transform supported by GDI+ is known as the world transform. It involves a traditional 3 × 3 matrix, but it’s possible to skip the matrix and use some very handy methods instead. To begin, let’s look at this program that displays the first paragraph of Herman Melville’s *Moby-Dick*.

**MobyDick.cs**

```csharp
// MobyDick.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MobyDick: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new MobyDick());
    }
    public MobyDick()
    {
        Text = "Moby-Dick by Herman Melville";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        // Insert RotateTransform, ScaleTransform, TranslateTransform, and other calls here.
        grfx.DrawString("Call me Ishmael. Some years ago\nnever " + "mind how long precisely\nhaving little " + "or no money in my purse, and nothing " + "particular to interest me on shore, I " + "thought I would sail about a little and " + "see the watery part of the world. It is " + "a way I have of driving off the spleen, " + "and regulating the circulation. Whenever " + "I find myself growing grim about the " + "mouth; whenever it is a damp, drizzly " + "November in my soul; whenever I find " + "myself involuntarily pausing before " + "coffin warehouses, and bringing up the " + "rear of every funeral I meet; and " + "especially whenever my hypos get such an " + "upper hand of me, that it requires a " + "strong moral principle to prevent me " +
```
"from deliberately stepping into the " +
"street, and methodically knocking " +
"people's hats off then, I account it " +
"high time to get to sea as soon as I " +
"can. This is my substitute for pistol " +
"and ball. With a philosophical flourish " +
"Cato throws himself upon his sword; I " +
"quietly take to the ship. There is " +
"nothing surprising in this. If they but " +
"knew it, almost all men in their degree, " +
"some time or other, cherish very nearly " +
"the same feelings towards the ocean with " +
"me."

Font, new SolidBrush(clr),
new Rectangle(0, 0, cx, cy));

This is nothing you haven't seen before except that I've indicated where you can add a line or two of code, recompile, and see what happens.

Here's the first one you can try:
grfx.RotateTransform(45);

Make sure you insert this call before the DrawString call. The effect is to rotate the text 45 degrees clockwise:

Simple enough, no? Notice that the text is still within the rectangle specified in the DrawString call, but that rectangle has been effectively rotated along with the text. You can also print it, but I must warn you that it may take quite some time for the print spooler file to be created.

What's affected by RotateTransform? All the Graphics drawing functions covered so far.

The argument to RotateTransform is a float value, and it can be positive or negative. Try this:
grfx.RotateTransform(-45);
The text is rotated 45º counterclockwise. The angle can also be greater than 360º or less than -360º.

For our particular example, any value that doesn't resolve to an angle between -90º and 90º will cause the text to be rotated right off the visible area of the window.

Successive calls to RotateTransform are cumulative. The calls
grfx.RotateTransform(5);
grfx.RotateTransform(10);
grfx.RotateTransform(-20);
result in the text being rotated 5º counterclockwise.

Now try this:
grfx.ScaleTransform(1, 3);
This function increases the coordinates and sizes of displayed graphics. The first argument affects the horizontal coordinates and sizes, and the second argument affects the vertical. In the MobyDick program, calling this function causes the width of the text to be the same but makes the font characters three times taller. The call
grfx.ScaleTransform(3, 1);
doesn't affect the height of the characters but makes them three times wider. The display rectangle increases likewise, so the text has the same line breaks. You can also combine the two effects:
grfx.ScaleTransform(3, 3);
Again, these are float values and they are compounded. Scaling both the horizontal and vertical sizes by a factor of 3 can be accomplished by the following two calls:
grfx.ScaleTransform(3, 1);
grfx.ScaleTransform(1, 3);
Or by
But what you'll probably find most interesting is that blowing up the text doesn't make it all jaggy. It looks as if you are using a different sized font rather than increasing the size of an existing font.

Can the scale values be negative? Yes, they can. However, if you try it now, you won't see anything. But what you probably find most interesting is blowing up the text doesn't make it all jaggy. It looks as if you are using a different sized font rather than increasing the size of an existing font.

But shifting the text helps demonstrate other techniques. I want you to insert the call:
grfx.TranslateTransform(100, 50);

That will begin the text in the center of the client area or printer page. That's not very interesting by itself, but now insert the call:
grfx.TranslateTransform(cx / 2, cy / 2);

That will begin the text in the center of the client area or printer page. That's not very interesting by itself, but now insert the call:
grfx.TranslateTransform(cx / 2, cy / 2);

Now try this:

Now that is interesting, isn't it? What happens is that the text is reflected around the vertical axis, appearing as a mirror image in the bottom left quadrant of the client area.

But shifting the text helps demonstrate other techniques. I want you to insert the call:
grfx.TranslateTransform(100, 50);

That will begin the text in the center of the client area or printer page. That's not very interesting by itself, but now insert the call:
grfx.TranslateTransform(cx / 2, cy / 2);

That will begin the text in the center of the client area or printer page. That's not very interesting by itself, but now insert the call:
grfx.TranslateTransform(cx / 2, cy / 2);

Now try this:

Now that is interesting, isn't it? What happens is that the text is reflected around the vertical axis, appearing as a mirror image in the bottom left quadrant of the client area.

Now try this:

Now that is interesting, isn't it? What happens is that the text is reflected around the vertical axis, appearing as a mirror image in the bottom left quadrant of the client area.
Now replace that `ScaleTransform` call with this one:
```csharp
gfx.ScaleTransform(1, -1);
```
Now the text is reflected around the horizontal axis and appears upside down. Again, you can combine the two effects:
```csharp
gfx.ScaleTransform(-1, -1);
```
Now you know why you couldn't use the `ScaleTransform` call by itself with negative arguments—the text would be flipped off the visible surface of the client area. You need to move the text farther from the left and top edge to see the effect.

OK, now let's try switching around the order of the `TranslateTransform` and one of the `ScaleTransform` calls:
```csharp
gfx.ScaleTransform(-1, 1);
gfx.TranslateTransform(cx / 2, cy / 2);
```
Now you see nothing, and you probably figure that it's because the text has been somehow moved off the surface of the client area. There are two ways to bring it back. One way is to change the first argument of the `TranslateTransform` call so that it's negative:
```csharp
gfx.ScaleTransform(-1, 1);
gfx.TranslateTransform(-cx / 2, cy / 2);
```
Now it's back to being reflected around the vertical axis in the center of the client area. By the way, I'm not expecting you to understand why this works yet. Indeed, at this point, confusion would not be inappropriate.

To add to that confusion, here's another way to do it. Leave the first argument the way it was, but use this overload of the `TranslateTransform` method:
```csharp
gfx.ScaleTransform(-1, 1);
gfx.TranslateTransform(cx / 2, cy / 2, MatrixOrder.Append);
```
Each of the three methods we've looked at so far—`RotateTransform`, `ScaleTransform`, and `TranslateTransform`—is overloaded to allow a final `MatrixOrder` argument, which is an enumeration type.
defined in the `System.Drawing.Drawing2D` namespace. (That's why I've conveniently included the additional `using` statement at the top of the MobyDick program.)

Here are the formal definitions of the `Graphics` methods I've discussed in this section so far, plus another:

**Graphics Methods (selection)**

```csharp
void TranslateTransform(float dx, float dy)
void TranslateTransform(float dx, float dy, MatrixOrder mo)
void ScaleTransform(float sx, float sy)
void ScaleTransform(float sx, float sy, MatrixOrder mo)
void RotateTransform(float fAngle)
void RotateTransform(float fAngle, MatrixOrder mo)
void ResetTransform()
```

The `ResetTransform` call makes everything go back to normal. The `MatrixOrder` enumeration has just two members:

**MatrixOrder Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepend</td>
<td>0</td>
<td>Default</td>
</tr>
<tr>
<td>Append</td>
<td>1</td>
<td>Switches order of application</td>
</tr>
</tbody>
</table>

What these enumeration values do will become evident before the conclusion of the chapter.

**The Big Picture**

The coordinates you pass to the various drawing methods implemented in the `Graphics` class are said to be world coordinates. World coordinates are first subjected to the world transform, which is the thing we've been playing around with by calling `TranslateTransform`, `ScaleTransform`, and `RotateTransform`. I'll formalize the world transform shortly.

The world transform converts world coordinates to page coordinates. The page transform—the transform defined by the `PageUnit` and `PageScale` properties of the `Graphics` object—converts those page coordinates to device coordinates, which are pixels relative to the upper left corner of the client area or printable area of the printer page.

For functions such as the `Graphics` method `MeasureString` or the `Font` method `GetHeight`, this process is reversed. Device coordinates are converted to page coordinates, which are then converted to world coordinates and returned by the method.

**Linear Transforms**

Let's look at the mathematical effect of calling the various transform methods. The simplest of these methods seems to be

```csharp
grfx.TranslateTransform(dx, dy);
```

where I'm symbolizing the arguments with `dx` and `dy`. (The `d` stands for `delta`, mathematically meaning `change`.) The world transform that results from this method call is

\[
x_{\text{page}} = x_{\text{world}} + dx
\]

\[
y_{\text{page}} = y_{\text{world}} + dy
\]
Easy enough. As you saw, the `TranslateTransform` call resulted in all coordinates being offset.

Similarly, here's a call to `ScaleTransform`:

```csharp
grfx.ScaleTransform(s_x, s_y);
```

The `s` stands for `scale`. This world transform involves not an addition but a multiplication:

\[
\begin{align*}
   x_{\text{page}} &= s_x \cdot x_{\text{world}} \\
   y_{\text{page}} &= s_y \cdot y_{\text{world}} 
\end{align*}
\]

This scaling effect is very similar to the page transform.

When you call

```csharp
grfx.RotateTransform(\alpha);
```

with an angle of \(\alpha\), well, I won't make you guess. The resultant transform is obviously a bit more complicated and looks like this:

\[
\begin{align*}
   x_{\text{page}} &= x_{\text{world}} \cdot \cos(\alpha) + y_{\text{world}} \cdot \sin(\alpha) \\
   y_{\text{page}} &= -x_{\text{world}} \cdot \sin(\alpha) + y_{\text{world}} \cdot \cos(\alpha) 
\end{align*}
\]

This little table of sines and cosines may help you verify that these formulas do indeed work:

<table>
<thead>
<tr>
<th>Angle (\alpha)</th>
<th>Sine</th>
<th>Cosine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>(\sqrt{2})</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>90</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>135</td>
<td>(\sqrt{2})</td>
<td>(-\sqrt{2})</td>
</tr>
<tr>
<td>180</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>225</td>
<td>(-\sqrt{2})</td>
<td>(-\sqrt{2})</td>
</tr>
<tr>
<td>270</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>315</td>
<td>(-\sqrt{2})</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>360</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

By the way, if you’re familiar with this stuff from experience with other graphics programming environments, the two formulas for rotations might look slightly off. That's because GDI+ expresses rotations clockwise. In more mathematically oriented environments, rotations are counterclockwise. In that case, the sine term in the first formula is negative and the sine term in the second is positive.

We can generalize all three of these transforms into the two formulas

\[
\begin{align*}
   x_{\text{page}} &= s_x \cdot x_{\text{world}} + r_x \cdot y_{\text{world}} + d_x \\
   y_{\text{page}} &= r_y \cdot x_{\text{world}} + s_y \cdot y_{\text{world}} + d_y 
\end{align*}
\]

where \(s_x, s_y, r_x, r_y, d_x,\) and \(d_y\) are constants that define the particular transform. You’ve already been introduced to the scaling factors \(s_x\) and \(s_y\), and the translation factors \(d_x\) and \(d_y\). You’ve also seen that certain specific combinations of \(s_x, s_y, r_x,\) and \(r_y\)—combinations defined by trigonometric
functions of particular angles—can result in rotation. The \( r_x \) and \( r_y \) factors also have a meaning in themselves, and the graphical effect of these two constants will soon become apparent.

These two formulas taken together are known as the general linear transformation of the plane. Although \( x_\text{page} \) and \( y_\text{page} \) are functions of both \( x_\text{world} \) and \( y_\text{world} \), these formulas don't involve powers of \( x_\text{world} \) or \( y_\text{world} \) or anything like that. That the world transform is linear implies certain restrictions on what you can do with the world transform.

- The world transform will always transform a straight line into another straight line. Straight lines will never become curved.
- A pair of parallel lines will never be transformed into lines that are not parallel.
- Two objects equal in size to each other will never be transformed into two objects unequal in size.
- Parallelograms (including rectangles) will always be transformed into other parallelograms; ellipses will always be transformed into other ellipses.

When you start off with a new, clean Graphics class on entry to a Paint or a PrintPage event, the world transform in effect is called the identity transform: the \( s_x \) and \( s_y \) factors are set equal to 1; the other factors are set to 0. The ResetTransform method restores the Graphics object to the identity transform.

As you've seen, the effects of successive calls to TranslateTransform, ScaleTransform, and RotateTransform are accumulated. However, the resultant world transform differs depending on the order that you call these methods. It's fairly easy to demonstrate why. This won't be pretty, so it's OK if you cover your eyes during the scary parts.

First, let's assume we have one world transform that I'll call \( T_1 \):

\[
\begin{align*}
  x' &= s_{x_1} \cdot x + r_{x_1} \cdot y + d_{x_1} \\
  y' &= r_{y_1} \cdot x + s_{y_1} \cdot y + d_{y_1}
\end{align*}
\]

Rather than using subscripts indicating world coordinates and page coordinates, the world coordinates are simply \( x \) and \( y \), and the page coordinates are \( x' \) and \( y' \). Let's assume a second transform called \( T_2 \) with different factors:

\[
\begin{align*}
  x' &= s_{x_2} \cdot x + r_{x_2} \cdot y + d_{x_2} \\
  y' &= r_{y_2} \cdot x + s_{y_2} \cdot y + d_{y_2}
\end{align*}
\]

Applying \( T_1 \) first to world coordinates and then \( T_2 \) to the result produces this transform:

\[
\begin{align*}
  x' &= (s_{x_2} \cdot s_{x_1}) \cdot x + (s_{x_2} \cdot r_{x_1}) \cdot y + (s_{x_2} \cdot d_{x_1} + r_{x_2} \cdot r_{y_1}) \cdot x + (r_{x_2} \cdot s_{y_1} \cdot y + r_{x_2} \cdot d_{y_1}) + d_{x_2} \\
  y' &= (r_{y_2} \cdot s_{y_1} + s_{y_2} \cdot r_{y_1}) \cdot x + (r_{y_2} \cdot r_{x_1} + s_{y_2} \cdot s_{y_1}) \cdot y + (r_{y_2} \cdot d_{x_1} + s_{y_2} \cdot d_{y_1} + d_{y_2})
\end{align*}
\]

Consolidating the terms, you arrive at this:

\[
\begin{align*}
  x' &= (s_{x_2} \cdot s_{x_1} + s_{x_2} \cdot r_{y_1}) \cdot x + (s_{x_2} \cdot r_{y_1} + r_{x_2} \cdot s_{y_1}) \cdot y + (s_{x_2} \cdot d_{x_1} + r_{x_2} \cdot d_{y_1} + d_{x_2}) \\
  y' &= (r_{y_2} \cdot s_{x_1} + s_{y_2} \cdot r_{y_1}) \cdot x + (r_{y_2} \cdot r_{x_1} + s_{y_2} \cdot s_{y_1}) \cdot y + (r_{y_2} \cdot d_{x_1} + s_{y_2} \cdot d_{y_1} + d_{y_2})
\end{align*}
\]

If you apply \( T_2 \) first and then \( T_1 \), you get something different:

\[
\begin{align*}
  x' &= s_{x_1} \cdot s_{x_2} \cdot x + s_{x_1} \cdot r_{y_2} \cdot y + s_{x_1} \cdot d_{x_2} + r_{x_1} \cdot r_{x_2} \cdot x + r_{x_1} \cdot s_{y_2} \cdot y + r_{x_1} \cdot d_{y_2} + d_{x_1} \\
  y' &= r_{y_1} \cdot s_{x_2} \cdot x + r_{y_1} \cdot r_{y_2} \cdot y + r_{y_1} \cdot d_{x_2} + s_{y_1} \cdot r_{x_2} \cdot x + s_{y_1} \cdot s_{y_2} \cdot y + s_{y_1} \cdot d_{y_2} + d_{y_1}
\end{align*}
\]

Consolidating the terms, you obtain

\[
\begin{align*}
  x' &= (s_{x_1} \cdot s_{x_2} + r_{x_1} \cdot r_{y_2}) \cdot x + (s_{x_1} \cdot r_{y_2} + s_{x_1} \cdot s_{x_2}) \cdot y + (s_{x_1} \cdot d_{x_2} + r_{x_1} \cdot d_{y_2} + d_{x_1}) \\
  y' &= (r_{y_1} \cdot s_{y_2} + s_{y_1} \cdot r_{y_2}) \cdot x + (r_{y_1} \cdot r_{y_2} + s_{y_1} \cdot s_{y_2}) \cdot y + (r_{y_1} \cdot d_{x_2} + s_{y_1} \cdot d_{y_2} + d_{y_1})
\end{align*}
\]

And that, my friends, is why you get different results depending on whether you call ScaleTransform or TranslateTransform first.

---


**Introducing Matrixes**
When something is very messy in mathematics (like the calculations I just demonstrated), the solution usually doesn't involve removing something but introducing something new. Here it will be very useful to introduce a matrix, particularly because the mathematics of matrix algebra are well known (at least to mathematicians). You can represent a linear transform by a matrix; applying multiple transforms is equivalent to multiplying the matrices.

A matrix is a rectangular array of numbers. Here's an array with three columns and two rows:

\[
\begin{bmatrix}
27 & 9 & 14 \\
3 & 0 & 88
\end{bmatrix}
\]

Arrays are usually symbolized by capital letters. When multiplying two matrices like so:

\[
A \times B = C
\]

the number of columns in A must be the same as the number of rows in B. The number of rows in the product C is equal to the number of rows in A. The number of columns in C is equal to the number of columns in B. The number in the i\text{th} row and j\text{th} column in C is equal to the sum of the products of the numbers in the i\text{th} row of A times the corresponding numbers in the j\text{th} column of B.

Matrix multiplication is not commutative. The product \(A \times B\) does not necessarily equal the product \(B \times A\).

If we weren't dealing with translation, we could represent the world coordinates \((x, y)\) as a 1 \(\times\) 2 matrix and the transformation matrix as a 2 \(\times\) 2 matrix. You multiply these two matrices and express the resultant page coordinates \((x', y')\) as another 1 \(\times\) 2 matrix:

\[
\begin{bmatrix}
x & y
\end{bmatrix} \times \begin{bmatrix}
s_x & r_y \\
r_x & s_y
\end{bmatrix} = \begin{bmatrix}
x' & y'
\end{bmatrix}
\]

Applying the multiplication rules to the matrices gives us the formulas

\[
x' = s_x \cdot x + r_x \cdot y \\
y' = r_y \cdot x + s_y \cdot y
\]

These formulas are not quite complete, however. The world transform also involves a translation factor. To get the matrix multiplication to work right, the world coordinates and page coordinates must be expanded to 1 \(\times\) 3 matrices, and the transform is a 3 \(\times\) 3 matrix:

\[
\begin{bmatrix}
x & y & 1
\end{bmatrix} \times \begin{bmatrix}
s_x & r_y & 0 \\
r_x & s_y & 0 \\
d_x & d_y & 1
\end{bmatrix} = \begin{bmatrix}
x' & y' & 1
\end{bmatrix}
\]

Here are the resultant formulas:

\[
x' = s_x \cdot x + r_x \cdot y + d_x \\
y' = r_y \cdot x + s_y \cdot y + d_y
\]

The type of transform that can be represented by a matrix like this is often called a matrix transform.

The matrix transform that doesn't do anything has scaling factors of 1, and \(r\) and \(d\) have factors of 0:
This is called the identity matrix.

See Pettofrezzo, section 1-2 for examples.

**The Matrix Class**

The matrix transform is encapsulated in the Matrix class defined in the System.Drawing.Drawing2D namespace. You can create a Matrix object using one of four constructors, two of which are shown here:

### Matrix Constructors (selection)

- `Matrix()`
- `Matrix(float sx, float ry, float rx, float sy, float dx, float dy)`

The second constructor allows you to specify all six constants that define the matrix transform. The scaling factors $sx$ and $sy$ must be nonzero! (If they're not, you'll get an exception error.)

The Graphics class has a read/write property named Transform that is a Matrix object:

### Graphics Property (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>Transform</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Whenever you call the TranslateTransform, ScaleTransform, RotateTransform, or ResetTransform method, the Transform property is affected. You can also set the Transform property directly. The call

```csharp
grfx.Transform = new Matrix(1, 0, 0, 1, 0, 0);
```

has the same effect as ResetTransform.

The Matrix class has five properties, all of which are read-only:

### Matrix Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float[]</td>
<td>Elements</td>
<td>get</td>
<td>Six transformation constants</td>
</tr>
<tr>
<td>float</td>
<td>OffsetX</td>
<td>get</td>
<td>Transform $d_x$ constant</td>
</tr>
<tr>
<td>float</td>
<td>OffsetY</td>
<td>get</td>
<td>Transform $d_y$ constant</td>
</tr>
<tr>
<td>bool</td>
<td>IsIdentity</td>
<td>get</td>
<td>Diagonal of 1's</td>
</tr>
<tr>
<td>bool</td>
<td>IsInvertible</td>
<td>get</td>
<td>Can be inverted</td>
</tr>
</tbody>
</table>

Let's now look at an example of compounded transforms. Suppose you first call

```csharp
grfx.ScaleTransform(2, 2);
```

Your program could then examine the resultant matrix by calling

```csharp
float[] afElements = grfx.Transform.Elements;
```

You'll see the values 2, 0, 0, 2, 0, 0, which can be represented as the following matrix:
Now you call

```
grfx.TranslateTransform(100, 100);
```

By itself, that would result in the matrix

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
100 & 100 & 1
\end{bmatrix}
\]

However, the new transform is a composite of the two method calls. The matrix representing the second call is multiplied by the existing `Transform` property, and the result is the new `Transform` property:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
100 & 100 & 1
\end{bmatrix} \times \begin{bmatrix}
2 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 1
\end{bmatrix} = \begin{bmatrix}
2 & 0 & 0 \\
0 & 2 & 0 \\
200 & 200 & 1
\end{bmatrix}
\]

Now try making the `ScaleTransform` and `TranslateTransform` calls in the opposite order:

```
grfx.TranslateTransform(100, 100);
grfx.ScaleTransform(2, 2);
```

Again, the resultant transform is calculated by multiplying the second matrix by the first matrix:

\[
\begin{bmatrix}
2 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
100 & 100 & 1
\end{bmatrix} = \begin{bmatrix}
2 & 0 & 0 \\
0 & 2 & 0 \\
100 & 100 & 1
\end{bmatrix}
\]

You can also obtain this transform by calling

```
grfx.ScaleTransform(2, 2);
grfx.TranslateTransform(100, 100, MatrixOrder.Append);
```

The `MatrixOrder.Append` argument indicates that the new transform is to be appended to the existing transform. The default is `MatrixOrder.Prepend`.

The `Graphics` class has one more world transform method:

```
Graphics Methods (selection)
void MultiplyTransform(Matrix mat)
void MultiplyTransform(Matrix mat, MatrixOrder mo)
```

This method lets you multiply the existing transform matrix by a new one.
I’ll discuss the Matrix class more in Chapter 15.

Shear and Shear Alike

Let’s go back to the MobyDick program and insert the following statement:

```csharp
grfx.Transform = new Matrix(1, 0, 0, 3, 0, 0);
```

This statement has the same effect as the call

```csharp
grfx.ScaleTransform(1, 3);
```

What we haven’t experimented with yet are the rx and ry factors used by themselves. Consider the following call:

```csharp
grfx.Transform = new Matrix(1, 0, 0.5f, 1, 0, 0);
```

This call results in the following transform matrix:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0.5 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

And these are the transform formulas:

\[x' = x + 0.5 \cdot y\]
\[y' = y\]

Notice that the \(x\) coordinate values are increased by the \(y\) value. When \(y\) equals 0 (the top of the client area), no transform will occur. As \(y\) gets larger going down the client area, \(x\) is increased likewise. The result is an effect called shear.

Specifically, the effect here is called horizontal shear, or \(x\)-shear. Unfortunately, the word shear starts with the same letter as scale, so to identify the shear factors in the transform formulas, I’ve used the last letter of shear.

You can also set the vertical shear, or \(y\)-shear, factor like so:

```csharp
grfx.Transform = new Matrix(1, 0.5f, 0, 1, 0, 0);
```

This matrix is
The transform formulas are

\[ x' = x \]
\[ y' = 0.5 \cdot x + y \]

Notice that each line of text still begins at the left margin of the client area:

Rotation is actually a combination of horizontal shear and vertical shear. However, some combinations, like this one, won't work:

```csharp
grfx.Transform = new Matrix(1, 1, 1, 1, 0, 0);
```

This defines the transform

\[ x' = x + y \]
\[ y' = x + y \]

This transform would cause an image to be compressed into a single line. It will generate an exception before it does that. But this call works:

```csharp
grfx.Transform = new Matrix(1, 1, -1, 1, 0, 0);
```

It results in the following display:
If you set the first four arguments to the square root of 1/2,
grfx.Transform = new Matrix(0.707f, 0.707f, -0.707f, 0.707f, 0, 0);
you'll produce the same result as the call we started this whole exploration with:
grfx.RotateTransform(45);

**Combining Transforms**

In theory, you don't need the page transform at all. All the page transform does is scaling, and you can certainly do that and a lot more with the world transform. It's often convenient to combine the two transforms, however, particularly if you're interested in drawing figures of a particular size that are then subjected to the world transform.

This program draws 36 one-inch squares that are rotated around the center of the display area.

**RotatedRectangles.cs**

//------------------------------------------------
// RotatedRectangles.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RotatedRectangles: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new RotatedRectangles());
    }
    public RotatedRectangles()
    {
        Text = "Rotated Rectangles";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
Pen    pen   = new Pen(clr);
grfx.PageUnit  = GraphicsUnit.Pixel;
PointF[] aptf  = { (PointF) grfx.VisibleClipBounds.Size }; 
grfx.PageUnit  = GraphicsUnit.Inch;
grfx.PageScale = 0.01f;

grfx.TransformPoints(CoordinateSpace.Page, 
                   CoordinateSpace.Device, aptf);

grfx.TranslateTransform(aptf[0].X / 2, aptf[0].Y / 2);

for (int i = 0; i < 36; i++)
{
    grfx.DrawRectangle(pen, 0, 0, 100, 100);
    grfx.RotateTransform(10);
}

The hard part here is calculating the arguments to the TranslateTransform call, which is necessary to shift the world coordinate origin to the center of the display area. The OnPage method changes the page unit to pixels in order to get the VisibleClipBounds property in units of pixels. OnPage then switches to a page unit of 1/100 inch and transforms the display area width and height to page coordinates. The TranslateTransform call uses these values halved.

The for loop is the easy part: It draws a rectangle 100 units wide and high positioned at the point (0, 0). The RotateTransform call then rotates $10^\circ$ in preparation for the next iteration. And here's what it looks like:

Knowing how to rotate objects around an origin will come in handy in the analog clock program in Chapter 10.
Chapter 8: Taming the Mouse

Overview

United States patent number 3,541,541, filed June 21, 1967, describes an "X-Y Position Indicator for a Display System."[1] The inventor is listed as Douglas C. Engelbart of the Stanford Research Institute (SRI). The word mouse is never mentioned in the original patent, of course, but it's obvious that's what the patent describes.

Doug Engelbart (born 1925) founded the Augmentation Research Center at SRI to advance computer hardware and software in pursuit of an ambitious goal: to create tools for the augmentation of human intelligence. As Engelbart recollected in 1986, "We wanted to start experimenting with screen selection. The idea of working and interacting very actively with the display meant that we had to tell the computer what we were looking at, so we needed a screen selection device. There was a lot of argument about light pens and tracking balls in those days, but none of those arguments served our needs very directly. I wanted to find the best thing that would serve us in the context in which we wanted to work—text and structured items and interactive commands…. I dug up some notes of mine describing a possibility that turned into the very first mouse."[2]

By 1972, the mouse had found its way to the Xerox Palo Alto Research Center (PARC), where it became part of the Alto, the machine that is commonly regarded as the first implementation of a graphical user interface and the precursor to the personal computer. But it wasn't until the 1983 introduction of the ill-fated Apple Lisa and the more successful Macintosh a year later that the mouse started to become a common accessory on every well-dressed computer.

While the keyboard is adequate for alphanumerical input and rudimentary cursor movement, the mouse provides a more intimate connection between the user and objects on the screen. As an extension of the user's fingers, the mouse can point, grab, and move. The mouse has also adapted itself well to new types of applications: although games players and graphics artists were among the first to experience the mouse, in more recent years, the mouse has proved invaluable in navigating through hypertext-oriented mediums like the Web.[1] U.S. patents are available for viewing at http://www.uspto.gov/patft. You'll need a TIFF viewer for patents issued prior to 1976.[2]


The Dark Side of the Mouse

When Microsoft Windows was first released in 1985, the mouse was still a relatively rare appliance in the IBM-compatible world. The early developers of Windows felt that users shouldn't be required to buy a mouse in order to use the product. The mouse was made an optional accessory for Windows, and keyboard alternatives to the mouse were provided in all the little programs that came with Windows. (This is still the case: check out the help information in the Windows Calculator to see how each button is industriously assigned a keyboard alternative.) Third-party software developers were also encouraged to follow Microsoft's lead and provide keyboard interfaces in their own applications. Although the mouse has become a nearly universal PC peripheral, part of the legacy of Windows involves an openness to mouseless system configurations. When at all possible, I still like the idea of providing keyboard equivalents for mouse actions. Touch typists in particular prefer keeping their hands on the keyboard, and I suppose everyone has had the experience of "losing" a mouse on a cluttered desk or having a mouse too clogged up with mouse gunk to work well. The keyboard equivalents usually don't cost much in terms of thought or effort, and they can deliver more functionality to users who prefer them.

There are a number of strong reasons why keyboard alternatives to the mouse must be considered an essential part of any Windows application, most having to do with accessibility. For example, as the average age of computer users increases, some people—myself included—have suffered from painful and debilitating injuries to their hands, arms, and shoulders that are a direct result of excessive mouse use. Sometimes these problems can even be traced to a single application. I've
made an extra effort in searching out keyboard alternatives in the applications I use, and it's
disheartening to find applications whose developers have seemingly given up on providing a well-
rounded user interface.

**Ignoring the Mouse**

Since Chapter 2, you've been writing and running programs that respond to mouse input. The
standard form includes a mouse interface that lets the user move the form around the screen by
dragging its caption bar, resize the form by dragging its sizing border, open the control box (also
known as the system menu) to select items, and trigger the minimize, maximize, and close buttons.
All this happens without any effort by you, the programmer. Obviously, Windows is handling that
mouse input itself.

As you learned in Chapter 4, it's not necessary for a Windows Forms program to worry about mouse
input when it implements a scroll bar. The scroll bar code itself handles the mouse input and
responds appropriately.

Beginning in Chapter 12, I'll start talking about the many predefined controls available in Windows
Forms. Later chapters will cover menus and dialog boxes. You'll discover that all these user interface
enhancements handle their own mouse input. Indeed, that's the primary purpose of controls: to
encapsulate a low-level interface to the keyboard and mouse, and to provide a higher-level interface
that you as a programmer can deal with.

This chapter involves those times when you need to directly handle mouse input within your client
area, which, of course, is something that not all applications need to do. Those programmers who
will adorn their client areas with predefined controls may never need to deal directly with mouse
input. However, if you ever want to write your own controls, having a solid foundation in mouse
handling is a necessity.

**Some Quick Definitions**

A mouse is a pointing device with one or more buttons. The mouse is the object that sits on your
desk. When you move the mouse, the Windows environment moves a small bitmap image called the
mouse cursor on the screen. (In some graphical environments—and even in some of the Windows
Forms documentation—the mouse cursor is referred to as a pointer.)

The mouse cursor has a hot spot that corresponds to a precise pixel location on the screen. For
example, the hot spot of the default arrow cursor is the tip of the arrow. This is what is meant by the
location of the mouse cursor. I hope you won't be too alarmed if I'm occasionally a little sloppy and
refer to the location or position of the mouse rather than the mouse cursor. Rest assured that the
mouse is still on your desk and not crawling up your screen.

Clicking the mouse is pressing and releasing a mouse button. Dragging the mouse is holding down
the mouse button and moving the mouse. Double-clicking is clicking the mouse button twice in
succession. For an action to qualify as a double-click, both clicks must occur within a set period of
time and with the mouse cursor in approximately the same location on the screen. If you ever need
to know these values (and it's unlikely you will), the SystemInformation class contains two static
read-only properties with this information:

<table>
<thead>
<tr>
<th>SystemInformation Static Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>Size</td>
</tr>
</tbody>
</table>

The user has control over these settings using the Mouse item in Control Panel.

**Information About the Mouse**

Can you run your computer without a mouse? Well, why don't you try? Shut down your computer,
unplug the mouse, restart, and see what happens. Windows 2000 doesn't seem to complain.
Ctrl+Esc (or the Windows key on some keyboards) brings up the Start menu, and you can navigate
through your programs, documents, or favorites list with the keyboard cursor-movement keys.
A Windows Forms program may want to determine whether a mouse is present and, if so, how many buttons it has. Again, the `SystemInformation` class comes to the rescue:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>MousePresent</td>
<td>get</td>
<td>Indicates whether a mouse is installed</td>
</tr>
<tr>
<td>int</td>
<td>MouseButtons</td>
<td>get</td>
<td>Indicates the number of buttons on the mouse</td>
</tr>
<tr>
<td>bool</td>
<td>MouseButtonsSwapped</td>
<td>get</td>
<td>Indicates whether buttons are swapped</td>
</tr>
</tbody>
</table>

`MousePresent` returns `true` if a mouse is installed, and `MouseButtons` indicates the number of buttons on the mouse. If a mouse is installed, the number of buttons could be reported as one, two, three, four, or five, with two and three buttons probably being the most common on machines currently running Windows.

The `MouseButtonsSwapped` property returns `true` if the user has used the Mouse item on Control Panel to swap the functionality of the left and right mouse buttons. This swapping is usually done by left-handed users who put the mouse on the left side of the keyboard and want to use the forefinger of the left hand to carry out the most common mouse operations.

You don't normally need to know about button swapping. However, if you want to write a computer-based training program that includes an animation that shows mouse buttons being pressed, you might want to delight the user by showing the mouse in the configuration that the user has selected. (Of course, nothing prevents a user from moving the mouse to the left of the keyboard without swapping the buttons—a technique I've used deliberately to lessen my mouse use—or swapping the buttons and using the right hand.)

Regardless of any button swapping, the button called the left button is really the primary button. This is the button that carries out the most common activities of selecting items, dragging icons, and triggering actions.

The right mouse button has come to be used for invoking context menus. These are menus that appear at the mouse cursor position and pertain to options that apply only to the area where the mouse cursor is currently located. For example, in Internet Explorer, if the cursor is positioned over an image and you press the right mouse button, you get several options, including one to save the picture to a file. If the mouse isn't positioned over a picture but on some other part of the page, you won't have an option to save the image, but you will have an option to print the page. I'll discuss how you can create context menus in Chapter 14.

**The Mouse Wheel**

"Build a better mousetrap and the world will beat a path to your door," my mother used to tell me, unknowingly paraphrasing Ralph Waldo Emerson. Nowadays, it might make more sense to build a better mouse.

The three-button mouse never achieved much popularity under Windows until Microsoft introduced the IntelliMouse. While not exactly intelligent in the conventional sense, the IntelliMouse does include an enhancement in the form of a little wheel between the two buttons. If you press on this wheel, it functions as a third mouse button (referred to in programming interfaces as the middle button). But you can also rotate the wheel with your finger, and wheel-aware programs can respond by scrolling or zooming a document.

As gimmicky as this may sound, it turns out that the mouse wheel is habit-forming, particularly for reading long documents or Web pages. The big advantage is that you don't need to keep the mouse cursor positioned over the scroll bar; it can be anywhere within the document.

Once again, `SystemInformation` is the place to go for information about the mouse wheel:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>MouseWheelPresent</td>
<td>get</td>
<td>Returns <code>true</code> if wheel is present</td>
</tr>
</tbody>
</table>
The mouse wheel doesn’t turn smoothly but instead has a definite notched, or clicked, feel. To ensure that applications respond to the mouse wheel consistently, each notch (called a detent in the .NET Framework documentation) is supposed to correspond to a certain number of text lines that the application scrolls through the document. The MouseWheelScrollLines property indicates that number of lines. For the Microsoft IntelliMouse, the property currently returns 3. However, future super-duper mouse gizmos may have a finer notch, and in that case, MouseWheelScrollLines might someday return 2 or 1.

If you think it might be interesting to add mouse wheel support to supplement the scroll bar in one of the SysInfo programs we developed in Chapter 4, don’t bother. The scroll bars created by the auto-scroll facility respond to the mouse wheel automatically.

Or maybe not. The full quotation “If a man can write a better book, preach a better sermon, or make a better mousetrap than his neighbor, though he builds his house in the woods the world will make a beaten path to his door” is attributed to a lecture by Emerson but doesn’t appear in his writings. See Bartlett's Familiar Quotations, 16th ed. (Boston: Little, Brown, 1992), 430. It's also widely acknowledged these days that this charming sentiment just ain’t so. A good marketing strategy is also necessary for the commercial success of mousetraps or any other consumer item.

The Four Basic Mouse Events

Mouse activity is communicated to a Windows Forms application in the form of events. The Control class defines nine mouse events and nine corresponding protected methods; any class descended from Control (including Form) also inherits these nine methods.

Although a detailed discussion of controls awaits us in Chapter 12, it’s helpful to get an idea of how the mouse works with controls. So for now, imagine a form or a dialog box covered with controls such as buttons, text labels, text-entry fields, and so forth. These controls are considered children of the form. Likewise, the form is known as the parent of the controls. We’ve already had contact with this notion in the SysInfoPanel program in Chapter 4 when the Parent property of the Panel control was assigned to the Form object.

Only one control receives any particular mouse event. A control receives mouse events only when it is both enabled and visible, that is, when both the Enabled and Visible properties are set to true. Usually, mouse events are received only by the control directly underneath the mouse cursor.

If a child control is enabled and visible, and you pass the mouse cursor over the control, the child control receives the mouse events rather than the parent. If the child control is either disabled or invisible, the parent receives the mouse events. It’s as if the child were transparent. If multiple controls are stacked on the same physical point, the enabled and visible control highest in the Z-order receives the mouse events, that is, visibly on top of all other overlapping controls. I’ll explain this concept in Chapter 12.

Any object derived or instantiated from Form receives mouse events only when the mouse is positioned over the form’s client area; the Form object does not receive mouse events when the cursor is positioned over the form’s border, caption bar, control box, minimize box, maximize box, close box, menu, or scroll bars.

However, as you’ll see, under some circumstances a control or form receives mouse events when the mouse cursor is not positioned over the control. This feature is known as mouse capturing, and it assists forms and controls in tracking mouse movement. I’ll have much more to say on this subject later in this chapter.

Here are the four basic mouse events:
### Control Events (selection)

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseDown</td>
<td>OnMouseDown</td>
<td>MouseEventArgs</td>
<td>MouseEventArgs</td>
</tr>
<tr>
<td>MouseUp</td>
<td>OnMouseUp</td>
<td>MouseEventArgs</td>
<td>MouseEventArgs</td>
</tr>
<tr>
<td>MouseMove</td>
<td>OnMouseMove</td>
<td>MouseEventArgs</td>
<td>MouseEventArgs</td>
</tr>
<tr>
<td>MouseWheel</td>
<td>OnMouseWheel</td>
<td>MouseEventArgs</td>
<td>MouseEventArgs</td>
</tr>
</tbody>
</table>

As the names imply, the `MouseDown` and `MouseUp` events indicate a button being pressed or released. The `MouseMove` event signals mouse movement, and the `MouseWheel` event occurs when the user rolls the mouse wheel.

These four events are the only events associated with objects of type ` MouseEventArgs`. The ` MouseEventArgs` class has five read-only properties:

<table>
<thead>
<tr>
<th><code> MouseEventArgs</code> Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>MouseButtons</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

`X` and `Y` are integers that indicate the position of the mouse cursor hot spot in pixels relative to the upper left corner of the client area. These two properties are valid for all four mouse events.

The `Button` property indicates the mouse button or buttons involved in the event. This property isn't valid for `MouseWheel` events. The `Button` property is a `MouseButtons` enumeration value:

<table>
<thead>
<tr>
<th><code> MouseButtons</code> Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
</tr>
<tr>
<td>Middle</td>
</tr>
<tr>
<td>XButton1</td>
</tr>
<tr>
<td>XButton2</td>
</tr>
</tbody>
</table>

In this enumeration, the word `Left` should be interpreted as the user's primary mouse button—the button that invokes application menus and lets the user resize and move forms. The `Right` mouse button is the button that invokes context menus. `XButton1` and `XButton2` refer to buttons in the IntelliMouse Explorer, which has five buttons.

For `MouseDown` and `MouseUp` events, the `Button` property indicates the particular button being pressed or released.

For `MouseMove` events, the `Button` property indicates which button or buttons are currently pressed. Notice that the values are bit flags that can be combined. For example, if both the left and right buttons are pressed, the `Button` property equals 0x00300000. If the ` MouseEventArgs` object is named `mea`, the following expression returns true if the right mouse button and only the right mouse button is pressed:

```
(mea.Button == MouseButtons.Right)
```
The following expression is true if the right mouse button is pressed, regardless of the other mouse buttons:

```csharp
(mea.Button & MouseButtons.Right != 0)
```

The Clicks property is valid only for MouseDown events and is normally set to 1. The property is set to 2 if the MouseDown event follows a previous MouseDown event quickly enough to qualify as a double-click.

The Delta property is valid only for MouseWheel events. If you roll the wheel forward one click, the Delta property will typically equal 120, and if you roll it back one click, the Delta property will typically equal −120.

**Doing the Wheel**

Let's get the mouse wheel out of the way first so we can focus on more conventional aspects of mouse use. In the previous paragraph, I mentioned the value 120. This is a rare instance of a number essential to Windows Forms programming—or at least the processing of mouse wheel events—that is not associated with a static property or an enumeration value. In the Win32 header files, an identifier named WHEEL_DELTA is defined as 120; in Windows Forms programs that use the mouse wheel, you'll have to hard-code this value or define your own `const` variable.

When you get a MouseWheel event, you calculate the number of text lines to scroll like so:

```csharp
mea.Delta * SystemInformation.MouseWheelScrollLines / 120
```

Currently, this calculation yields either 3 or −3, but including the SystemInformation constant in the calculation allows your program to adapt better to future mouse wheel devices that have finer wheel gradations. Positive values indicate that the user is pushing the wheel forward; the program should respond by scrolling toward the top of the document. Negative values mean that the user is pulling the wheel back, and the program should scroll toward the bottom of the document.

The following program demonstrates the use of the mouse wheel by displaying (and scrolling) Edgar Allan Poe's creepy poem "Annabel Lee."

**PoePoem.cs**

```csharp
//--------------------------------------
// PoePoem.cs © 2001 by Charles Petzold
//--------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PoePoem: Form
{
    const string strAnnabelLee =

        "It was many and many a year ago,\n" +
        " In a kingdom by the sea,\n" +
        "That a maiden there lived whom you may know\n" +
        " By the name of Annabel Lee;\x2014\n" +
        "And this maiden she lived with no other thought\n" +
        " Than to love and be loved by me.\n" +
        "\n" +
        "I was a child and she was a child\n" +
        " In this kingdom by the sea,\n" +
        "But we loved with a love that was more than love\x2014\n" +
```
I and my Annabel Lee

With a love that the wingèd seraphs of Heaven

Coveted her and me.

And this was the reason that, long ago,

In this kingdom by the sea,

A wind blew out of a cloud, chilling

My beautiful Annabel Lee;

So that her highborn kinsmen came

And bore her away from me,

To shut her up in a sepulchre,

In this kingdom by the sea.

The angels, not half so happy in Heaven,

Went envying her and me

Yes! that was the reason (as all men know,

In this kingdom by the sea)

That the wind came out of the cloud by night,

Chilling and killing my Annabel Lee.

But our love it was stronger by far than the love

Of those who were older than we

Of many far wiser than we

And neither the angels in Heaven above

Nor the demons down under the sea

Can ever dissever my soul from the soul

Of the beautiful Annabel Lee:

For the moon never beams, without bringing me dreams

Of the beautiful Annabel Lee;

And the stars never rise, but I feel the bright eyes

Of the beautiful Annabel Lee:

And so, all the night-tide, I lie down by the side

Of my darling my darling my life and my bride,

In her sepulchre there by the sea

In her tomb by the sounding sea.

[May 1849]
// See whether the program makes sense.

if (!SystemInformation.MouseWheelPresent)
{
    MessageBox.Show("Program needs a mouse with a mouse wheel!",
                    "PoePoem", MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
    return;
}
// Otherwise go normally.

Application.Run(new PoePoem());

public PoePoem()
{
    Text = """Annabel Lee" by Edgar Allan Poe";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;
    ResizeRedraw = true;

    // Calculate the number of lines in the text.

    int iIndex = 0;

    while((iIndex = strAnnabelLee.IndexOf('
', iIndex)) != -1)
    {
        iTextLines++;
        iIndex++;
    }

    // Obtain line-spacing value.

    Graphics grfx = CreateGraphics();
    cyText = Font.GetHeight(grfx);
    grfx.Dispose();

    OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    iClientLines = (int) (ClientSize.Height / cyText);

    iStartLine = Math.Max(0,
                          Math.Min(iStartLine, iTextLines - iClientLines));
protected override void OnMouseWheel(MouseEventArgs mea)
{
    int iScroll =
    mea.Delta * SystemInformation.MouseWheelScrollLines / 120;

    iStartLine -= iScroll;
    iStartLine = Math.Max(0,
                          Math.Min(iStartLine, iTextLines - iClientLines));
    Invalidate();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    grfx.DrawString(strAnnabelLee, Font, new SolidBrush(ForeColor),
                    0, -iStartLine * cyText);
}

Notice that the program checks whether a mouse wheel is installed and lets the user know if it can't find one. I've put this check in Main, but that's not the only option in programs that shouldn't run under certain conditions. You can alternatively override the OnLoad method of the Form class and check at that time. The Load event occurs after the constructor code executes but before the form is made visible on the screen. In that case, if the program determines that it shouldn't run, it can display a message box and call Close to prevent the form from being displayed. Where you cannot abort the display of a form is in the form's constructor. Neither Close nor the static Application.Exit method works there.

The text of the poem includes embedded line feed characters and is stored in a string variable. The program counts the number of lines during the form's constructor and saves the result in a field named iTextLines. The constructor also obtains the text line spacing by calling the GetHeight method of the form's Font property. The return value is stored in a field named cyText.

The remainder of the initialization occurs in the OnResize method. The constructor must first call OnResize explicitly. Thereafter, OnResize is called whenever the user resizes the form. OnResize uses cyText to calculate iClientLines, the number of lines that can fit in the client area.

The iStartLine variable is the line of text that should appear at the top of the client area. It is initialized at zero. The OnMouseWheel method adjusts the value using the calculation I showed earlier.

Sometimes programs that scroll text are written so that scrolling all the way to the bottom makes the last line of text appear at the top of the client area. But it's not necessary to allow the user to scroll quite that far. All that's required is for the last line of text to be visible at the bottom of the client area. For this reason, both the OnMouseWheel and OnResize methods in PoePoem include a calculation using the Math.Min and Math.Max methods. This calculation ensures that iStartLine is non-negative and also that it's based on the amount of text that can fit in the client area. If you make the client area tall enough to fit the entire text, the text won't scroll at all.

Mouse Movement

Let's next look at the MouseMove event. This program is called MouseWeb but it has nothing to do with the World Wide Web; instead, it overrides the OnMouseMove method to draw a web that connects the current mouse position with the corners and sides of the client area.
using System;
using System.Drawing;
using System.Windows.Forms;

class MouseWeb: Form
{
    Point ptMouse = Point.Empty;

    public static void Main()
    {
        Application.Run(new MouseWeb());
    }

    public MouseWeb()
    {
        Text = "Mouse Web";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }

    protected override void OnMouseMove(MouseEventArgs mea)
    {
        Graphics grfx = CreateGraphics();

        DrawWeb(grfx, BackColor, ptMouse);
        ptMouse = new Point(mea.X, mea.Y);
        DrawWeb(grfx, ForeColor, ptMouse);

        grfx.Dispose();
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        DrawWeb(pea.Graphics, ForeColor, ptMouse);
    }

    void DrawWeb(Graphics grfx, Color clr, Point pt)
    {
        int cx = ClientSize.Width;
        int cy = ClientSize.Height;
        Pen pen = new Pen(clr);

        grfx.DrawLine(pen, pt, new Point(         0,          0));
        grfx.DrawLine(pen, pt, new Point(    cx / 4,          0));
        grfx.DrawLine(pen, pt, new Point(    cx / 2,          0));
    }
}
Move the mouse cursor within the client area, and the center of the web follows. A typical screen looks like this:

![Mouse Web](image)

The program displays the web first during the `OnPaint` method using a `Point` structure stored as a field and initialized to (0, 0). During the `OnMouseMove` method, the program erases the previous figure by redrawing it using the background color. The program then redraws the web based on the new mouse position using the foreground color.

Notice how the program stops responding to the mouse as soon as the mouse cursor leaves the client area. Even if the mouse cursor is moved over the program’s caption bar, the calls to `OnMouseMove` stop.

Or do they? Try this: Move the mouse cursor to `MouseWeb`’s client area. The center of the web follows the mouse as usual. Now press one of the mouse buttons. With the button still pressed, move the mouse cursor outside the client area. The center of the web continues to follow the cursor! Release the mouse button. The program stops responding. This is a feature called mouse capture, and it’s an important part of the technique of tracking the mouse position.

### Tracking and Capturing the Mouse

When a program needs to draw something or move something in response to mouse movement, it uses a technique called mouse tracking. Most often, mouse tracking begins when a mouse button is
pressed and ends when the button is released. A program written for an environment not supporting event handling would probably track the mouse by sitting in a `while` loop continuously monitoring the mouse cursor position. A Windows Forms program, however, must track the mouse by responding to events. This architecture forces the programmer to approach the exercise as if dealing with a state machine.

Here's a fun little program that demonstrates some rudimentary mouse cursor tracking.

**MouseConnect.cs**

```csharp
// ------------------------------------------
// MouseConnect.cs © 2001 by Charles Petzold
// ------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class MouseConnect : Form
{
    const int iMaxPoints = 1000;
    int iNumPoints = 0;
    Point[] apoint = new Point[iMaxPoints];

    public static void Main()
    {
        Application.Run(new MouseConnect());
    }
    public MouseConnect()
    {
        Text = "Mouse Connect: Press, drag quickly, release";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ClientSize += ClientSize; // Double the client area.
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
        {
            iNumPoints = 0;
            Invalidate();
        }
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
        {
            apoint[iNumPoints++] = new Point(mea.X, mea.Y);
            Graphics grfx = CreateGraphics();
        }
    }
```
To use this program, you press the left mouse button anywhere in the client area, move the mouse cursor quickly around, and then release the button. For every OnMouseMove call the program gets, it stores the X and Y properties of the MouseEventArgs object and draws a tiny mark at that point.

When you release the button, the OnMouseUp method invalidates the client area. OnPaint responds by connecting every point to every other point, sometimes creating a big blob and sometimes making an interesting pattern:

As you can see, as I whipped the mouse cursor around, I twice drifted outside the client area. The program didn’t seem to mind. It connected all the lines, even those with points outside the client area. The lines are clipped to the client area, but all the points are still correctly stored. If you create
such an image and make the client area a bit taller, you'll see the bottom of the figure. You can even release the mouse button outside of MouseConnect's client area and the program will work normally.

This is probably what you want to happen: the user is signaling a desire to work with MouseConnect by pressing the mouse button within its client area, and this activity should end only when the user releases the mouse button—regardless of where the mouse cursor is or has been.

Whenever you press any mouse button on a control or in a form's client area, the control or form captures the mouse and forces each subsequent mouse event to be sent to itself. The capture ends when the user releases the mouse button. Mouse capture capability is virtually a prerequisite for tracking the mouse, and it is automatically provided for you. A bool property of the Control class indicates when the mouse is captured:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Capture</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Although this property is writable, you can't arbitrarily set it. In particular, you can't force a mouse capture when a mouse button isn't pressed. However, you can cancel mouse capture at any time by setting the property to false. (I'll do that later in this chapter.) The property is also useful for informational purposes. The property is true during both the MouseDown event that begins mouse capture andMouseMove events when the mouse is captured, and false during the MouseUp event that releases mouse capture. The mouse isn't automatically captured on the second click of a double-click.

### Adventures in Tracking

Generally, it's fairly easy to write some mouse-tracking code that works 99.5 percent of the time. This program is quite similar in structure to MouseConnect but it does something much more conventional, which is letting you drag the mouse to draw a rectangle.

**BlockOut.cs**

```csharp
//---------------------------------------
// BlockOut.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BlockOut: Form
{
    bool bBlocking, bValidBox;
    Point ptBeg, ptEnd;
    Rectangle rectBox;

    public static void Main()
    {
        Application.Run(new BlockOut());
    }
    public BlockOut()
    {
        Text = "Blockout Rectangle with Mouse";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
}
```
protected override void OnMouseDown(MouseEventArgs mea)
{
    if (mea.Button == MouseButtons.Left)
    {
        ptBeg = ptEnd = new Point(mea.X, mea.Y);
        Graphics grfx = CreateGraphics();
        grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
        grfx.Dispose();
        bBlocking = true;
    }
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    if (bBlocking)
    {
        Graphics grfx = CreateGraphics();
        grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
        ptEnd = new Point(mea.X, mea.Y);
        grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
        grfx.Dispose();
        Invalidate();
    }
}

protected override void OnMouseUp(MouseEventArgs mea)
{
    if (bBlocking && mea.Button == MouseButtons.Left)
    {
        Graphics grfx = CreateGraphics();
        rectBox = Rect(ptBeg, new Point(mea.X, mea.Y));
        grfx.DrawRectangle(new Pen(ForeColor), rectBox);
        grfx.Dispose();
        bBlocking = false;
        bValidBox = true;
        Invalidate();
    }
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    if (bValidBox)
To use this program, you press the left mouse button, drag, and release. As you're dragging, the program draws a rectangle outline. When you release the button, the program fills in the rectangle. If you want, you can then define a new rectangle that replaces the first one.

BlockOut uses two bool variables stored as fields: bBlocking and bValidBox. The bBlocking variable indicates that the user is blocking out a rectangle. It is set to true during the OnMouseDown method and false during OnMouseUp. The OnMouseMove method tests the variable to determine what it should do. If bBlocking is true, OnMouseMove erases the previous rectangle outline by drawing it in the background color and draws a new rectangle outline in the foreground color. When you release the button, the OnMouseUp method sets bBlocking to false and bValidBox to true. This latter variable allows the OnPaint method to draw the filled rectangle.

Customarily, during the OnMouseMove method, I would use a technique called exclusive-OR (or XOR) drawing. XOR drawing is a technique that doesn't merely write colored pixels out to the display device but instead reverses the colors of the existing pixels. An XOR line drawn on a black background appears white, an XOR line drawn on a white background appears black, and an XOR line drawn on a cyan background appears red. The advantage of this technique is that a second XOR line of the same coordinates erases the first.

GDI+ doesn't support exclusive-OR drawing, however, which is why I'm forced to erase the previous rectangle by using the background color in the OnMouseMove method. When you're blocking out a new rectangle over an existing filled rectangle, some unsightly artifacts are created, and these must be cleaned up. That's why the OnMouseMove processing terminates with a call to Invalidate to generate a Paint event. That Invalidate call isn't strictly needed, but if you remove it, you'll see why I felt compelled to include it. With XOR drawing, the Invalidate call wouldn't be necessary at all.

The omission of XOR drawing is certainly a flaw in GDI+, but BlockOut has some flaws of its own.

If you experiment a little, you'll see that the BlockOut program works just fine most of the time. Because the mouse is captured when you press the mouse button, you can move the mouse cursor outside the client area and the program will continue to get OnMouseMove calls. You can also release the mouse button outside the client area and the program will get a call to OnMouseUp.

But try this: While blocking out a rectangle with the left button held down, press and release the right button. When the right button is released, the form loses the mouse capture. It will now respond to mouse movement only when the mouse cursor is within the form's client area. Now move the mouse cursor outside the client area and release the left button. The form doesn't get a call to OnMouseUp because the mouse is no longer captured. Now move the mouse (with no buttons pressed) back inside the client area. The program responds to mouse movement even though no mouse button is pressed! This behavior is clearly undesirable.

A few fixes that help solve these problems are apparent:

- Tracking should be terminated whenever any button is released. This approach more closely mimics the way mouse capture is lost.
OnMouseMove processing should include a check that the left button is still pressed. If a form loses the mouse capture, it's probably better for a rectangle outline to lie dormant rather than for the program to respond to mouse movement with no button pressed.

Pressing the Esc key should terminate mouse tracking.

Here's a better version of the program.

BetterBlockOut.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterBlockOut : Form
{
    bool bBlocking, bValidBox;
    Point ptBeg, ptEnd;
    Rectangle rectBox;

    public static void Main()
    {
        Application.Run(new BetterBlockOut());
    }
    public BetterBlockOut()
    {
        Text = "Better Blockout";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
        {
            ptBeg = ptEnd = new Point(mea.X, mea.Y);
            Graphics grfx = CreateGraphics();
            grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
            grfx.Dispose();
            bBlocking = true;
        }
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        if (bBlocking && (mea.Button & MouseButtons.Left) != 0)
        {
        }
    }
}
Graphics grfx = CreateGraphics();
grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
ptEnd = new Point(mea.X, mea.Y);
grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
grfx.Dispose();
Invalidate();
}

protected override void OnMouseUp(MouseEventArgs mea)
{
    if (bBlocking)
    {
        Graphics grfx = CreateGraphics();
        rectBox = Rect(ptBeg, new Point(mea.X, mea.Y));
grfx.DrawRectangle(new Pen(ForeColor), rectBox);
grfx.Dispose();

        bBlocking = false;
bValidBox = true;
        Invalidate();
    }
}

protected override void OnKeyPress(KeyPressEventArgs kpea)
{
    if (bBlocking && kpea.KeyChar == '\x001B')   // Escape
    {
        Graphics grfx = CreateGraphics();
grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
grfx.Dispose();

        bBlocking = false;
        Invalidate();
    }
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    if (bValidBox)
grfx.FillRectangle(new SolidBrush(ForeColor), rectBox);

    if (bBlocking)
grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
}

Rectangle Rect(Point ptBeg, Point ptEnd)
In some situations, this program can still lose the mouse capture and not be aware of it. If you're in the middle of a tracking operation and you press Ctrl+Esc to bring up the Start menu or you press Alt+Tab to switch to another program, both BlockOut and BetterBlockOut will lose the mouse capture and not even know it. Losing the mouse capture doesn't have to be the result of something weird that a user does. Suppose you're in the middle of a tracking operation and a message box pops up complaining that the printer has run out of paper. That occurrence also causes the program to lose the mouse capture because the message box needs to respond to mouse input.

Wouldn't it be nice if there were an event to tell a form when it's lost the mouse capture? Well, if we were writing a Win32 program, we'd be able to trap the **WM_CAPTURECHANGED** message. This message occurs whenever a window is losing the mouse capture, whether normally (when the mouse button is released) or abnormally. Is it possible to implement a handler for this message in a Windows Forms program?

Yes, it is, and to do it you make use of the *NativeWindow* class. Here's a program that demonstrates how to use that class to implement an *OnLostCapture* method in a class derived from *Form*.

**CaptureLoss.cs**

```csharp
//------------------------------------------------------------------------------
// CaptureLoss.cs © 2001 by Charles Petzold
//------------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CaptureLoss: Form
{
    public static void Main()
    {
        Application.Run(new CaptureLoss());
    }
    public CaptureLoss()
    {
        Text = "Capture Loss";

        // Hook up NativeWindow object.

        CaptureLossWindow win = new CaptureLossWindow();
        win.form = this;
        win.AssignHandle(Handle);
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
    }
}
```
```csharp
Invalidate();
}
public void OnLostCapture()
{
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    if (Capture)
        grfx.FillRectangle(Brushes.Red, ClientRectangle);
    else
        grfx.FillRectangle(Brushes.Gray, ClientRectangle);
}
}

class CaptureLossWindow: NativeWindow
{
    public CaptureLoss form;

    protected override void WndProc(ref Message message)
    {
        if (message.Msg == 533) // WM_CAPTURECHANGED
            form.OnLostCapture();

        base.WndProc(ref message);
    }
}

I have two classes here: CaptureLoss is derived from Form, and CaptureLossWindow is derived from NativeWindow. I've added a field to CaptureLossWindow that is an object of type CaptureLoss. CaptureLossWindow also overrides the WndProc ("window procedure") method in NativeWindow. Win32 API programmers will recognize WndProc as the all-important function in every Windows application program that processes messages to the window that the program creates.

CaptureLoss creates an object of type CaptureLossWindow in its constructor. It assigns the field named form (the field I added to the class) to the form that the program creates. The constructor also calls the AssignHandle method implemented in NativeWindow. This call assigns to the CaptureLossWindow object the window handle associated with the form. The CaptureLossWindow then receives, through its WndProc method, all messages to the form. Win32 API programmers will be well familiar with these messages. The Message parameter to WndProc is a structure in System.Windows.Forms that contains properties corresponding to all the Win32 message parameters. When WndProc receives a message ID of 533 (which is WM_CAPTURECHANGED), it calls the OnLostCapture method of the CaptureLoss object.

The CaptureLoss class invalidates its client area when the OnMouseDown and OnLostCapture methods are called. The OnPaint method displays a red client area when the Capture property is true and a gray client area otherwise. (You can get a better view of mouse capturing if you enable the Show Window Contents While Dragging option in the Effects tab of the Display Properties dialog box in Control Panel. You'll notice if you grab the title bar of the window and move it partially off screen and then back, the uncovered area of the client area will be colored red until you release the
mouse button, at which point the entire client area turns gray again. If you make the window larger with the mouse, likewise the new area of the client area will be red, returning to gray when you release the mouse button. This happens because the *Capture* property is *true* even if the mouse button is pressed on the program's caption bar or sizing border.)

**Generalizing Code with Interfaces**

I want to call your attention to a little structural flaw in the *CaptureLoss* program. It would be best if you could reuse the *CaptureLossWindow* class in other programs. However, the name of the class using the *CaptureLossWindow* class is hard-coded in the definition of the *CaptureLossWindow* field:

```csharp
public CaptureLoss form;
```

Why does that class name need to be hard-coded here? Because changing it to

```csharp
public Form form;
```

simply won't work, and the reason is that the class later makes this call:

```csharp
form.OnLostCapture();
```

This call would generate a compile-time error because *form* is defined as an object of type *Form*, and the *OnLostCapture* method isn't defined in the *Form* class. It's defined in the *CaptureLoss* class.

Is there a way out of this quandary? A good solution is to make use of the C# *interface*. An interface looks something like a class definition. It can contain methods, properties, and an indexer. However, the interface includes only *signatures* of these members. None of these members has bodies.

As you know, a class can inherit from another class, and a class that seemingly doesn't inherit from any class actually inherits from *Object*. A class can also inherit from one or more interfaces. When a class inherits from an interface, it must implement all the methods and properties defined in the interface. (Interfaces can also inherit from other interfaces, in which case a class that derives from the interface must also implement all the methods and properties defined in the base interfaces.) The interface is the closest thing C# has to multiple inheritance, but the only elements inherited from an interface are names of methods and properties, and not any actual code.

Interfaces help generalize classes because an interface name can be used instead of a class name or a structure name to define a variable. The class can then call methods or properties that are defined in the interface.

Here's a little program named *CaptureLossNotifyWindow* that contains a class of that name but also an interface named *ICaptureLossNotify*. Interface names by convention begin with a capital *I*.

*CaptureLossNotifyWindow.cs*

```csharp
//------------------------------------------------------
// CaptureLossNotifyWindow.cs © 2001 by Charles Petzold
//------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

interface ICaptureLossNotify
{
    void OnLostCapture();
}

class CaptureLossNotifyWindow: NativeWindow
{
    public ICaptureLossNotify control;
}
protected override void WndProc(ref Message message)
{
    if (message.Msg == 533) // WM_CAPTURECHANGED
        control.OnLostCapture();

    base.WndProc(ref message);
}

The interface defines a signature only for a method named OnLostCapture. Notice that the field in the CaptureLossNotifyWindow class named control is defined as an object of type ICaptureLossNotify:

    public ICaptureLossNotify control;

This fact tells the compiler that the OnLostCapture method may be called on the control variable. And, indeed, that's what the WndProc method does.

For a form to take advantage of the CaptureLossNotifyWindow class, it must indicate that it is derived from ICaptureLossNotify as well as Form. This in turn indicates that the class implements the OnLostCapture method. Here's a program that implements better block-out code by deriving from ICaptureLossNotify.

EvenBetterBlockOut.cs
//-------------------------------------------------
// EvenBetterBlockOut.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class EvenBetterBlockOut: Form, ICaptureLossNotify
{
    bool bBlocking, bValidBox;
    Point ptBeg, ptEnd;
    Rectangle rectBox;

    public static void Main()
    {
        Application.Run(new EvenBetterBlockOut());
    }
    public EvenBetterBlockOut()
    {
        Text = "Even Better Blockout";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        // Hook up native window object.

        CaptureLossNotifyWindow win = new CaptureLossNotifyWindow();
        win.control = this;
protected override void OnMouseDown(MouseEventArgs mea) {
    if (mea.Button == MouseButtons.Left) {
        ptBeg = ptEnd = new Point(mea.X, mea.Y);

        Graphics grfx = CreateGraphics();
        grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
        grfx.Dispose();

        bBlocking = true;
    }
}

protected override void OnMouseMove(MouseEventArgs mea) {
    if (bBlocking) {
        Graphics grfx = CreateGraphics();
        grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
        ptEnd = new Point(mea.X, mea.Y);
        grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
        grfx.Dispose();
        Invalidate();
    }
}

public void OnMouseUp(Point pt) {
    if (bBlocking) {
        Graphics grfx = CreateGraphics();
        rectBox = Rect(ptBeg, new Point(mea.X, mea.Y));
        grfx.DrawRectangle(new Pen(BackColor), rectBox);
        grfx.Dispose();

        bBlocking = false;
        bValidBox = true;
        Invalidate();
    }
}

protected override void OnKeyPress(KeyPressEventArgs kpea) {
    if (kpea.KeyChar == '\x001B')
        Capture = false;
public void OnLostCapture()
{
    if (bBlocking)
    {
        Graphics grfx = CreateGraphics();
        grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
        grfx.Dispose();

        bBlocking = false;
        Invalidate();
    }
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    if (bValidBox)
        grfx.FillRectangle(new SolidBrush(ForeColor), rectBox);

    if (bBlocking)
        grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
}

Rectangle Rect(Point ptBeg, Point ptEnd)
{
    return new Rectangle(Math.Min(ptBeg.X, ptEnd.X),
                          Math.Abs(ptEnd.X - ptBeg.X),
}

The program completes the tracking operation normally when it gets a call to OnMouseUp and aborts the tracking operation when it gets a call to OnLostCapture that wasn't preceded by OnMouseUp.

EvenBetterBlockOut finally accommodates all the ways in which a program can lose the mouse capture.

**Clicks and Double-Clicks**

Here are the two highest-level mouse events:

<table>
<thead>
<tr>
<th>Control Events (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Click</td>
</tr>
<tr>
<td>DoubleClick</td>
</tr>
</tbody>
</table>

Notice that the EventArgs argument doesn't give you any information specific to the mouse. It doesn't even tell you what button was clicked or double-clicked, or where the mouse was located.
The *Click* event occurs when any mouse button is pressed and released over a control or the client area of a form. The event occurs right before the corresponding *MouseUp* event. If you press the mouse button while the mouse cursor is positioned over one control and release the mouse button over another control, a *Click* event is not generated.

The *DoubleClick* event occurs when the mouse is clicked twice. The event occurs right before the second *MouseUp* event. However, the second button-down must occur within a certain period of time and within a certain distance of the first. Here's a typical sequence of events for a double-click:
- MouseDown (Clicks property set to 1)
- Click
- MouseUp
- MouseMove
- MouseDown (Clicks property set to 2)
- DoubleClick
- MouseUp
- MouseMove

I used the *OnClick* method to trigger printing in the PrintableForm program in Chapter 5. Obviously, I didn't need to know where the mouse cursor was located when the button was pressed, or even which button was pressed.

It's more common for a program to install *Click* and *DoubleClick* event handlers for controls that it has created. For example, handling a button control's *Click* event is the normal way for a form to determine when the button has been clicked, as you'll discover in Chapter 12. Buttons (and other controls) also generate *Click* events when the keyboard is used to press the button, so the *Click* event is a convenient consolidation of keyboard and mouse input.

### Mouse-Related Properties

Although the *Click* and *DoubleClick* events aren't delivered with a *MouseEventArgs* object that indicates the location of the mouse cursor, that doesn't mean the information isn't available. The *Control* class supports two read-only static properties that indicate the position of the mouse and which buttons are currently pressed:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>MousePosition</td>
<td>get</td>
<td>Returns the position of the mouse in screen coordinates</td>
</tr>
<tr>
<td>MouseButtons</td>
<td>MouseButtons</td>
<td>get</td>
<td>Returns which buttons are currently pressed</td>
</tr>
</tbody>
</table>

You can use these properties while processing any event. Because they are static properties, you can even use them in a class not descended from *Control*.

Keep in mind that the *X* and *Y* properties of *MouseEventArgs* indicate the mouse cursor location in client area coordinates, and the *Control.MousePosition* property gives the position in screen coordinates. You'll have to use *PointToClient* to convert screen coordinates to client area coordinates if that's what you need.

I introduced the static property *Control.ModifierKeys* in Chapter 6 because it pertains to the keyboard:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keys</td>
<td>ModifierKeys</td>
<td>get</td>
<td>Status of Shift, Ctrl, and Alt keys</td>
</tr>
</tbody>
</table>

However, as I mentioned at that time, this property is most often used when processing mouse events. For example, if you want to initiate an action when the user presses the left mouse button with the Shift key (and only the Shift key) pressed, the *OnMouseDown* processing might start like this:

```csharp
```
Entering, Leaving, Hovering

Here are the final three mouse events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseEnter</td>
<td>OnMouseEnter</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>MouseLeave</td>
<td>OnMouseLeave</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>MouseHover</td>
<td>OnMouseHover</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

The `MouseEnter` event announces to a control (or a client area of a form) that the mouse cursor has been moved on top of the control. The control may want to change its appearance in response to this event. The `MouseLeave` event tells the control that the mouse is no longer located on top of the control.

The `MouseHover` event occurs after the cursor has entered the control (or the client area) and has stopped moving. The `MouseHover` event occurs at most only once between `MouseEnter` and `MouseLeave` events.

Here’s a program that provides a visual indication of these three events. The client area is colored green following a call to `OnMouseEnter` and restored to the normal background color following the `OnMouseLeave` call. In response to `OnMouseHover`, the client area is colored red for 1/10 second.

`EnterLeave.cs`

```csharp
//-----------------------------------------
// EnterLeave.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class EnterLeave : Form
{
    bool bInside = false;

    public static void Main()
    {
        Application.Run(new EnterLeave());
    }
    public EnterLeave()
    {
        Text = "Enter/Leave";
    }
    protected override void OnMouseEnter(EventArgs ea)
    {
        bInside = true;
        Invalidate();
    }
    protected override void OnMouseLeave(EventArgs ea)
    {
    }
    protected override void OnMouseMove(EventArgs ea)
    {
```
bInside = false;
Invalidate();
}
protected override void OnMouseHover(EventArgs ea)
{
  Graphics grfx = CreateGraphics();

  grfx.Clear(Color.Red);
  System.Threading.Thread.Sleep(100);
  grfx.Clear(Color.Green);
  grfx.Dispose();
}
protected override void OnPaint(PaintEventArgs pea)
{
  Graphics grfx = pea.Graphics;

  grfx.Clear(bInside ? Color.Green : BackColor);
}

The Mouse Cursor

The mouse cursor is the little bitmap image you see on the screen that indicates the location of the mouse. As you know, the cursor can change appearance depending on its location. Often it's an arrow, but if you pass it over a sizing border of a form, it changes into a double-headed arrow. In a text-entry field, the cursor becomes a vertical I-beam.

The mouse cursor is an object of type Cursor, a class defined in the System.Windows.Forms namespace. I'll describe the Cursor class in more detail in Chapter 11. In most cases, the easiest way to get a cursor object is by using the Cursors class. (Notice the plural.) The Cursors class—also defined in the System.Windows.Forms namespace—consists solely of 28 static read-only properties that return predefined objects of type Cursor.

<table>
<thead>
<tr>
<th>Cursors Static Read-Only Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppStarting</td>
</tr>
<tr>
<td>Arrow</td>
</tr>
<tr>
<td>Cross</td>
</tr>
<tr>
<td>Default</td>
</tr>
<tr>
<td>Hand</td>
</tr>
<tr>
<td>Help</td>
</tr>
<tr>
<td>HSplit</td>
</tr>
<tr>
<td>IBeam</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>NoMove2D</td>
</tr>
<tr>
<td>NoMoveHoriz</td>
</tr>
<tr>
<td>NoMoveVert</td>
</tr>
<tr>
<td>PanEast</td>
</tr>
<tr>
<td>PanNE</td>
</tr>
</tbody>
</table>
Even if you obtain \textit{Cursor} objects only from the \textit{Cursors} class, there are still three static properties of the \textit{Cursor} class that are useful:

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textbf{Type} & \textbf{Property} & \textbf{Accessibility} \\
\hline
\textit{Cursor} & \textit{Current} & get/set \\
\hline
\textit{Point} & \textit{Position} & get/set \\
\hline
\textit{Rectangle} & \textit{Clip} & get/set \\
\hline
\end{tabular}
\end{center}

You’ll recall that the \textit{Control} class includes a static property named \textit{MousePosition}, but that property is read-only. You can’t use it to set the mouse cursor position. The \textit{Cursor.Position} property is writable as well, although it’s not common for applications to set the position of the mouse cursor. (The Beziers program in \textbf{Chapter 13} uses \textit{Cursor.Position} to set the cursor position.) The \textit{Cursor.Clip} property limits the movement of the mouse cursor to a specified rectangle. You can set this property only if the mouse is captured. The \textit{Position} and \textit{Clip} properties are both in screen coordinates, so you probably need to use \textit{PointToClient} after obtaining the properties or \textit{PointToScreen} before setting the properties.

You can also set the current mouse cursor by using the \textit{Cursor.Current} property. However, you might find that this approach doesn’t always work. But let me show you first a couple cases in which the \textit{Cursor.Current} property \textit{does} work.

As you know, programs that must perform lengthy jobs generally display a cursor shaped like an hourglass, which is the predefined \textit{Cursors.WaitCursor} object. A program can display the hourglass cursor using the statement
\begin{center}
\texttt{Cursor.Current = Cursors.WaitCursor;}
\end{center}

The program can then carry out the lengthy job it needs to do and afterward restore the arrow cursor by calling
\begin{center}
\texttt{Cursor.Current = Cursors.Arrow;}
\end{center}

However, if the user is running Windows without a mouse installed, the hourglass cursor won’t be visible. To display a mouse cursor regardless of whether or not a mouse is installed, a program can make use of the following two static methods of the \textit{Cursor} class:

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{Cursor Static Methods} \\
\hline
\texttt{void Show()}
\hline
\texttt{void Hide()}
\hline
\end{tabular}
\end{center}

You can think of the mouse cursor as having a \textit{show-count} variable associated with it. If a mouse is installed, this show-count variable is initially set to 1. If a mouse is not installed, the show-count is initially 0. The \textit{Cursor.Show} method increments the show-count; the \textit{Cursor.Hide} method decrements it. The mouse cursor is visible if the show-count is greater than 0 and hidden otherwise.

What this means is that an application must balance its calls to \textit{Cursor.Show} and \textit{Cursor.Hide}. If a program calls \texttt{Show} more than \texttt{Hide}, it risks leaving a visible mouse cursor on the screen when a mouse isn’t installed. If a program calls \texttt{Hide} more than \texttt{Show}, the mouse cursor is made invisible. Fortunately, this problem affects the mouse cursor only when the mouse is positioned over the errant application.

One program in this chapter that might spend a considerable amount of time in its \textit{OnPaint} method is MouseConnect. Here’s a program that subclasses the \textit{MouseConnect} class and displays an hourglass cursor during the \textit{OnPaint} processing.

\texttt{MouseConnectWaitCursor.cs}

// ---------
using System;
using System.Drawing;
using System.Windows.Forms;

class MouseConnectWaitCursor: MouseConnect
{
    public new static void Main()
    {
        Application.Run(new MouseConnectWaitCursor());
    }
    public MouseConnectWaitCursor()
    {
        Text = "Mouse Connect with Wait Cursor";
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Cursor.Current = Cursors.WaitCursor;
        Cursor.Show();
        base.OnPaint(pea);
        Cursor.Hide();
        Cursor.Current = Cursors.Arrow;
    }
}

In this particular case, the calls to Show and Hide methods aren't necessary because if the mouse weren't installed, the user couldn't have initiated a long OnPaint call to begin with!

This next program uses a call to Cursor.Current during the OnMouseMove call to let you see what all 28 predefined cursors look like.

MouseCursors.cs

using System;
using System.Drawing;
using System.Windows.Forms;

class MouseCursors: Form
{
    Cursor[] acursor =
    {
        Cursors.AppStarting, Cursors.Arrow,     Cursors.Cross,
        Cursors.Default,     Cursors.Hand,       Cursors.Help,
        }
Cursors.HSplit,      Cursors.IBeam,       Cursors.No,
Cursors.NoMove2D,    Cursors.NoMoveHoriz, Cursors.NoMoveVert,
Cursors.PanEast,     Cursors.PanNE,       Cursors.PanNorth,
Cursors.PanNW,       Cursors.PanSE,       Cursors.PanSouth,
Cursors.PanSW,       Cursors.PanWest,     Cursors.SizeAll,
Cursors.SizeNESW,    Cursors.SizeNS,      Cursors.SizeNWSE,
Cursors.SizeWE,      Cursors.UpArrow,     Cursors.VSplit,
Cursors.WaitCursor
);
string[] astrCursor =
{
    "AppStarting",       "Arrow",             "Cross",
    "Default",           "Hand",              "Help",
    "HSplit",            "IBeam",             "No",
    "NoMove2D",          "NoMoveHoriz",       "NoMoveVert",
    "PanEast",           "PanNE",             "PanNorth",
    "PanNW",             "PanSE",             "PanSouth",
    "PanSW",             "PanWest",           "SizeAll",
    "SizeNESW",          "SizeNS",            "SizeNWSE",
    "SizeWE",            "UpArrow",           "VSplit",
    "WaitCursor"
};

public static void Main()
{
    Application.Run(new MouseCursors());
}

public MouseCursors()
{
    Text = "Mouse Cursors";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;
    ResizeRedraw = true;
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    int x = Math.Max(0, Math.Min(3, mea.X / (ClientSize.Width / 4)));
    int y = Math.Max(0, Math.Min(6, mea.Y / (ClientSize.Height / 7)));

    Cursor.Current = acursor[4 * y + x];
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics     grfx   = pea.Graphics;
Brush brush = new SolidBrush(ForeColor);
Pen pen = new Pen(ForeColor);
StringFormat strfmt = new StringFormat();

strfmt.LineAlignment = strfmt.Alignment = StringAlignment.Center;

for (int y = 0; y < 7; y++)
for (int x = 0; x < 4; x++)
{
  Rectangle rect = Rectangle.FromLTRB(
    x * ClientSize.Width / 4,
    y * ClientSize.Height / 7,
    (x + 1) * ClientSize.Width / 4,
    (y + 1) * ClientSize.Height / 7);

  grfx.DrawRectangle(pen, rect);
  grfx.DrawString(astrCursor[4 * y + x],
                  Font, brush, rect, strfmt);
}
}
}

The program displays a grid containing the names of the Cursors properties. Simply move the mouse cursor to one of the boxes to see what the cursor looks like. Here's a screen shot that shows the Cursors.Help cursor:

I mentioned earlier that Cursor.Current doesn't always work. It certainly works in these two programs, but they illustrate the only two ways in which Cursor.Current can be used. Here's the important rule: If your form (or any descendant of Control) does not set Cursor.Current during the MouseMove event, the mouse cursor will be set instead to the normal cursor associated with the form (or control) during that event. The MouseCursors program works because it sets Cursor.Current during the call to OnMouseMove.

What about MouseConnectWaitCursor? That one doesn't set Cursor.Current during OnMouseMove. But that program sets Cursor.Current during OnPaint and then resets the property before OnPaint is concluded. The program doesn't get OnMouseMove calls during that time. A method in a program is never interrupted to execute another method in the same thread.

What you can't do, however, is set Cursor.Current during a constructor or an OnMouseDown event or some other event and expect it to stick. As soon as the program gets a call to OnMouseMove, the cursor will be reset.
However, there is a way to set the cursor once and then forget about it. You assign a cursor to a control (or a form) by using the `Cursor` property defined in the `Control` class:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Cursor</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

For example, in a form's constructor, you can call

```csharp
Cursor = Cursors.Hand;
```

and the cursor will be a hand whenever you pass the mouse cursor over the form's client area. It will remain a hand until the program sets it to something else. I mentioned earlier that if you don't set `Cursor.Current` during the `OnMouseMove` method, the cursor is set to the normal cursor associated with the form or control. It's actually set to the value of the `Cursor` property. Somewhere behind the scenes, something equivalent to the following statement probably occurs during mouse movement before a Windows Forms program gets a call to `OnMouseMove`:

```csharp
Cursor.Current = Cursor;
```

That's the `Cursor` class on the left and the form's `Cursor` property on the right. The `Current` property of the `Cursor` class is static. The `Cursor` property of the `Control` class is not.

It makes sense that a control should have a cursor associated with it because some controls have different cursors. The most obvious examples are the text-entry controls `TextBox` and `RichTextBox`, which are associated with `Cursors.IBeam`.

Let's experiment with this technique by creating a program similar in functionality to MouseCursors. But instead of creating 28 boxes, I want to create 28 controls and assign each of them a different mouse cursor.

Back in Chapter 4, I created a `Panel` control in the SysInfoPanel program. I mentioned at the time that `Panel` controls were rather innocuous and didn't do much. But the panel suited our purposes in providing a surface for us on which to draw. The `Label` control is also a fairly benign control. The sole purpose of `Label` is to display some text. Here's a program that creates an array of 28 `Label` controls, each of which is assigned a different cursor.

```csharp
class MouseCursorsProperty.cs
//---------------------------------------------------
// MouseCursorsProperty.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class MouseCursorsProperty: Form
{
    Label[] acntl = new Label[28];

    public static void Main()
    {
        Application.Run(new MouseCursorsProperty());
    }
    public MouseCursorsProperty()
    {
        Cursor[] acursor =
```
Cursors.AppStarting, Cursors.Arrow,       Cursors.Cross,
Cursors.Default,       Cursors.Hand,        Cursors.Help,
Cursors.HSplit,        Cursors.IBeam,       Cursors.No,
Cursors.NoMove2D,      Cursors.NoMoveHoriz,
Cursors.NoMoveVert,    Cursors.PanEast,     Cursors.PanNE,       Cursors.PanNorth,
Cursors.PanNW,         Cursors.PanSE,       Cursors.PanSouth,
Cursors.PanSW,         Cursors.PanWest,      Cursors.SizeAll,
Cursors.SizeNESW,      Cursors.SizeNS,      Cursors.SizeNWSE,
Cursors.SizeWE,        Cursors.UpArrow,     Cursors.VSplit,
Cursors.WaitCursor

string[] astrCursor =
{
    "AppStarting",       "Arrow",             "Cross",
    "Default",           "Hand",              "Help",
    "HSplit",            "IBeam",             "No",
    "NoMove2D",          "NoMoveHoriz",       "NoMoveVert",
    "PanEast",           "PanNE",             "PanNorth",
    "PanNW",             "PanSE",             "PanSouth",
    "PanSW",             "PanWest",           "SizeAll",
    "SizeNESW",          "SizeNS",            "SizeNWSE",
    "SizeWE",            "UpArrow",           "VSplit",
    "WaitCursor"
};

Text = "Mouse Cursors Using Cursor Property";

for (int i = 0; i < 28; i++)
{
    acntl[i] = new Label();
    acntl[i].Parent = this;
    acntl[i].Text = astrCursor[i];
    acntl[i].Cursor = acursor[i];
    acntl[i].BorderStyle = BorderStyle.FixedSingle;
    OnResize(EventArgs.Empty);
}

protected override void OnResize(EventArgs ea)
{
    for (int i = 0; i < acntl.Length; i++)
    {
        acntl[i].Bounds = Rectangle.FromLTRB(
            (i % 4) * ClientSize.Width / 4,

The program creates the 28 Label objects during the form's constructor and saves them in an array named acntl, which is a field of the class. The constructor sets four properties of each Label object. The first of these four properties indicates that the parent of the control is the form, which means that the control will appear on the surface of the form's client area:

```csharp
acntl[i].Parent = this;
```

The Text property of the control is set to the name of one of the 28 predefined cursors:

```csharp
acntl[i].Text = astrCursor[i];
```

The program also sets the Cursor property of the control to the corresponding Cursor object:

```csharp
acntl[i].Cursor = acursor[i];
```

Finally, the BorderStyle property is set to a single line:

```csharp
acntl[i].BorderStyle = BorderStyle.FixedSingle;
```

What the constructor doesn't do is set the location and size of the control. The control's location and size are set during the form's OnResize method. Each control gets a size equal to 1/4 the width and 1/7 the height of the form's client area. (Well, not quite. Setting all the controls to the same width and height caused some rounding problems that resulted in gaps between the controls. The calculation of the controls' Bounds property using the Rectangle.FromLTRB method helps avoid that problem.)

The program looks similar to the early MouseCursors program. As you pass the mouse cursor over each control, the cursor changes automatically without any need to process the MouseMove event.

**An Exercise in Hit-Testing**

When you draw graphics figures or text on your form, you determine the coordinates of each item and call the appropriate methods to draw it. Often, however, a program uses a mouse interface to allow a user to point to and manipulate these items. That means that your program must work backward from the pointer coordinates to determine which graphical item the mouse is pointing to.

This process is called hit-testing, and it can tend to be quite complex, particularly if your client window contains figures that overlap or text in a variable-pitch font. But in some cases, hit-testing is fairly straightforward. In fact, the MouseCursors program shown earlier in this chapter used hit-testing to determine which mouse cursor to display.

The Checker program draws an array of boxes covering its client area. If you click one of these boxes with the mouse, the box is filled with an X. Click it again and the X disappears.

**Checker.cs**

```csharp
// Checker.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class Checker : Form
```
protected const int xNum = 5;       // Number of boxes horizontally
protected const int yNum = 4;       // Number of boxes vertically
protected bool[,] abChecked = new bool[yNum, xNum];
protected int cxBlock, cyBlock;

public static void Main()
{
    Application.Run(new Checker());
}

public Checker()
{
    Text = "Checker";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;
    ResizeRedraw = true;

    OnResize(EventArgs.Empty);
}

protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);            // Or else ResizeRedraw doesn't work

cxBlock = ClientSize.Width  / xNum;
    cyBlock = ClientSize.Height / yNum;
}

protected override void OnMouseUp(MouseEventArgs mea)
{
    int x = mea.X / cxBlock;
    int y = mea.Y / cyBlock;

    if (x < xNum && y < yNum)
    {
        abChecked[y, x] ^= true;
        Invalidate(new Rectangle(x * cxBlock, y * cyBlock,
                                 cxBlock, cyBlock));
    }
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Pen pen = new Pen(ForeColor);

    for (int y = 0; y < yNum; y++)
for (int x = 0; x < xNum; x++)
{
    grfx.DrawRectangle(pen, x * cxBlock, y * cyBlock,
                        cxBlock, cyBlock);
    if (abChecked[y, x])
    {
        grfx.DrawLine(pen, x * cxBlock, y * cyBlock,
                      (x + 1) * cxBlock, (y + 1) * cyBlock);
        grfx.DrawLine(pen, x * cxBlock, (y + 1) * cyBlock,
                      (x + 1) * cxBlock, y * cyBlock);
    }
}

Whenever the form is resized, the program recalculates \textit{cxBlock} and \textit{cyBlock} values, which indicate the size of each box. The program also maintains an array of \texttt{bool} values named \texttt{abChecked} that indicate whether a particular box is checked. The \texttt{OnPaint} method draws an outline around each box and, if \texttt{abChecked} for that box is \textit{true}, draws an X in the box.

The hit-testing occurs during the \texttt{OnMouseUp} method. (I chose \texttt{OnMouseUp} rather than \texttt{OnMouseDown} to more closely mimic \texttt{OnClick}, which occurs when the mouse button is released.) The program divides the mouse coordinates by \texttt{cxBlock} and \texttt{cyBlock} to get indices of \texttt{abChecked}. It then inverts the \texttt{bool} value and invalidates the corresponding rectangle.

Here's a typical Checker display after a few boxes have been checked:

![Checker Display](checker.png)

**Adding a Keyboard Interface**

I said at the outset of this chapter that you should try to write your Windows Forms programs so they are usable with either a mouse or the keyboard. I've been shamelessly ignoring my own rule to concentrate on mouse logic. However, I think this is a good opportunity to see what's involved in emulating the mouse with the keyboard in the Checker program.
The first decision you have to make is how the keyboard interface should work. For this program, a reasonable approach might be to let the user move the mouse cursor around the client area using the arrow keys and other cursor-movement keys. You could simulate a mouse click by using the Enter key or the spacebar.

This `CheckerWithKeyboard` class subclasses `Checker` and provides a keyboard interface.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckerWithKeyboard: Checker
{
    public new static void Main()
    {
        Application.Run(new CheckerWithKeyboard());
    }
    public CheckerWithKeyboard()
    {
        Text += " with Keyboard Interface";
    }
    protected override void OnGotFocus(EventArgs ea)
    {
        Cursor.Show();
    }
    protected override void OnLostFocus(EventArgs ea)
    {
        Cursor.Hide();
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        Point ptCursor = PointToClient(Cursor.Position);

        int x = Math.Max(0, Math.Min(xNum - 1, ptCursor.X / cxBlock));
        int y = Math.Max(0, Math.Min(yNum - 1, ptCursor.Y / cyBlock));

        switch(kea.KeyCode)
        {
            case Keys.Up:    y--;  break;
            case Keys.Down:  y++;  break;
            case Keys.Left:  x--;  break;
            case Keys.Right: x++;  break;
            case Keys.Home:  x = y = 0;     break;
        }
    }
}
```
case Keys.End:    x = xNum - 1; 
                  y = yNum - 1;  break;

case Keys.Enter:
    case Keys.Space:
        abChecked[y, x] ^= true;
        Invalidate(new Rectangle(x * cxBlock, y * cyBlock,
                                  cxBlock, cyBlock));
        return;

    default:
        return;
    }

x = (x + xNum) % xNum;

y = (y + yNum) % yNum;

Cursor.Position = PointToScreen(new Point(x*cxBlock + cxBlock/2,
                                           y*cyBlock +
                                           cyBlock/2));

Let's take a look at the OnKeyDown processing first. The program obtains the current cursor position by using Cursor.Position and converts the position to client area coordinates. The x and y variables indicate the row and column of the box that the cursor is closest to, where x ranges from 0 to one less than the number of boxes horizontally, and y ranges from 0 to one less than the number of boxes vertically.

For cursor-movement keys, the program modifies the x and y variables. The Home key moves the cursor to the upper left box; the End key moves it to the lower right box. For the Enter key or the spacebar, the program reacts as it does to OnMouseUp. It toggles the check-mark variable and invalidates the rectangle. The OnKeyDown processing concludes by calculating a new mouse cursor position and setting the Cursor.Position property.

By itself, such OnKeyDown processing would work fine except for one little problem: such a keyboard interface is most important if a mouse isn't installed. Yet, if a mouse isn't installed, the cursor isn't visible! That's why this program also overrides the OnGotFocus and OnLostFocus methods and simply calls Cursor.Show and Cursor.Hide.

Putting the Children to Work

Do the X marks in Checker remind you of anything? Perhaps very large versions of check boxes such as those seen in Windows dialog boxes? As we saw in the Mouse CursorsProperty program, controls can help you structure and modularize your programs, and they particularly help in hit-testing.

So far, I've demonstrated some simple uses of the Panel control and the Label control. But you can get even simpler than what I've shown. It's not necessary to use one of the predefined controls. You can create your own controls by subclassing the Control class. Control is the basis of all the predefined controls in Windows Forms. When you create your own controls, however, it's recommended that you derive from UserControl, which derives from Control by way of ScrollableControl and ContainerControl.

Here's a class derived from UserControl that has a single bool field, which it toggles in response to an OnClick call. During the OnPaint method, it draws a border around itself and, if the bool variable is set to true, an X.

CheckerChild.cs
```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckerChild : UserControl
{
    bool bChecked = false;

    public CheckerChild()
    {
        ResizeRedraw = true;
    }

    protected override void OnClick(EventArgs ea)
    {
        base.OnClick(ea);

        bChecked = !bChecked;
        Invalidate();
    }

    protected override void OnKeydown(KeyEventArgs kea)
    {
        switch (kea.KeyCode)
        {
            case Keys.Enter:
            case Keys.Space:
                OnClick(new EventArgs());
                break;
        }
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Pen pen = new Pen(ForeColor);

        grfx.DrawRectangle(pen, ClientRectangle);

        if (bChecked)
        {
            grfx.DrawLine(pen, 0, 0, ClientSize.Width,
                          ClientSize.Height);
            grfx.DrawLine(pen, 0, ClientSize.Height, ClientSize.Width,
                          0);
        }
    }
}
```
The class also responds to a depression of the Enter or spacebar key by simulating an `OnClick`
method call.

The following program creates 20 of these controls and puts them on the surface of the client area.
In this way, it's very similar to the MouseCursorsProperty program shown earlier except that it's
using this custom control rather than a `Label` control. The hard part is the `OnResize` call when the
form must change the `Location` and `Size` of each of the controls.

```csharp
class CheckerWithChildren: Form
{
    protected const int xNum = 5;
    protected const int yNum = 4;
    protected CheckerChild[,] acntlChild;

    public static void Main()
    {
        Application.Run(new CheckerWithChildren());
    }

    public CheckerWithChildren()
    {
        Text = "Checker With Children";
        BackColor = SystemColors.Window;
       ForeColor = SystemColors.WindowText;
        CreateChildren();

        OnResize(EventArgs.Empty);
    }

    protected virtual void CreateChildren()
    {
        acntlChild = new CheckerChild[yNum, xNum];

        for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
        {
            acntlChild[y, x] = new CheckerChild();
            acntlChild[y, x].Parent = this;
        }
    }
}
```
protected override void OnResize(EventArgs ea)
{
    int cxBlock = ClientSize.Width / xNum;
    int cyBlock = ClientSize.Height / yNum;

    for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
            {   
            acntlChild[y, x].Location = new Point(x*cxBlock,
                    y*cyBlock);
            acntlChild[y, x].Size = new Size(cxBlock, cyBlock);
            }
}  
}

Here's the really nice thing about this program: no hit-testing! The child control doesn't care where it gets clicked. If it gets a call to OnClick, it toggles the check mark. Windows itself is doing all the hit-testing by determining which control the mouse click should go to.

In this particular case, the parent form isn't interested in when the child gets clicked. But it could fairly easily install event handlers for the control's Click event. (The OnClick method in CheckerChild calls base.OnClick to ensure that Click event handlers are called.) The form would need a method defined like so:

void ChildOnClick(object obj, EventArgs ea)

And then in the loop when creating the controls, event handlers would be installed like so:

acntlChild[y, x].Click += new EventHandler(ChildOnClick);

The bChecked field of CheckerChild could be made public so that the parent could find the state of each child. Or CheckerChild could implement a property that provides access to the value.

Does the CheckerWithChildren program have a keyboard interface? Funny you should ask. If you run the program and press Enter or the spacebar, you'll toggle the X in the box in the upper left corner. Now press the Tab key seven times. Or the Right Arrow key, or the Down Arrow key, or any combination of these three keys seven times. Now press Enter or the spacebar. The box that gets toggled this time is the one in the middle of the second row.

Without any effort on the programmer's part, the form is responding to the Tab key and arrow keys by shifting the input focus among the 20 controls. The 20 controls were created in a particular order starting with the one in the upper left corner and then across each row, and then down to the next row. The Tab, Right Arrow, and Down Arrow keys shift the input focus to the next control in this order; the Left Arrow and Up Arrow keys and the Shift+Tab key combination shift the input focus to the previous control. You can also change the input focus by clicking on a control. This interface is implemented in the ContainerControl class, which is one of the ancestors of Form. The control itself responds to the Enter or spacebar key.

However, a common amenity is missing: the control isn't giving any indication when it has the input focus. Perhaps a wider border around the control would be sufficient.

Here's a new class that subclasses the original CheckerChild class to implement this feature.

CheckerChildWithFocus.cs
//----------------------------------------------------
// CheckerChildWithFocus.cs © 2001 by Charles Petzold
//----------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class CheckerChildWithFocus: CheckerChild
{
    protected override void OnGotFocus(EventArgs ea)
    {
        Invalidate();
    }

    protected override void OnLostFocus(EventArgs ea)
    {
        Invalidate();
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        base.OnPaint(pea);

        if (Focused)
        {
            Graphics grfx = pea.Graphics;
            grfx.DrawRectangle(new Pen(ForeColor, 5), ClientRectangle);
        }
    }
}

This new control invalidates itself when the control gains or loses focus, providing an opportunity for the OnPaint method to redraw its border based on the Focused property.

The form that uses these new controls is basically identical to CheckerWithChildren, but it needs to create children of type CheckerChildWithFocus rather than CheckerChild. For this reason, the new form overrides the CreateChildren method of the CheckerWithChildren class.

CheckerWithChildrenAndFocus.cs
//----------------------------------------------------------
// CheckerWithChildrenAndFocus.cs © 2001 by Charles Petzold
//----------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckerWithChildrenAndFocus: CheckerWithChildren
{
    public new static void Main()
    {
        Application.Run(new CheckerWithChildrenAndFocus());
    }

    public CheckerWithChildrenAndFocus()
    {  

Hit-Testing Text

I mentioned earlier in this chapter that one of the more complex hit-testing jobs involves text in a variable-pitch font. Basically, what you need to do is call MeasureString multiple times to attempt to figure out which character of displayed text (or, more accurately, which space between the characters) the user is clicking with the mouse.

In Chapter 6 I presented a program named TypeAway that showed how to echo keyboard input in a single line of displayed text. This program included a caret that the user moves with the arrow keys. Let's add a mouse interface to that program that sets the caret position based on the X property of the MouseEventArgs object passed as an argument to OnMouseDown. (For multiple lines of text, such a program would also use the Y property to determine the line of text the user was pointing to.)

HitTestText.cs

```
// HitTestText.cs © 2001 by Charles Petzold
//-------
using System;
using System.Drawing;
using System.Windows.Forms;

class HitTestText : TypeAway
{
    public new static void Main()
    {
        Application.Run(new HitTestText());
    }
    public HitTestText()
    {
        Text += " with Hit-Testing";
        Cursor = Cursors.IBeam;
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (strText.Length == 0)
            return;
```
Graphics grfx = CreateGraphics();
float xPrev = 0;
int i;

for (i = 0; i < strText.Length; i++)
{
    SizeF sizef = grfx.MeasureString(strText.Substring(0, i + 1),
        Font, Point.Empty,
        StringFormat.GenericTypographic);
        break;
    xPrev = sizef.Width;
}
iInsert = i;

grfx.Dispose();
PositionCaret();
}

Notice that the constructor sets the Cursor property of the form to Cursors.IBeam to make the program look like a real text editor.

The OnMouseDown method includes a for loop based on the number of characters in the stored text string. The comparison using calls to Math.Abs (absolute value) determines which space between the characters the X coordinate of the mouse cursor is closest to. It then sets the iInsert field to that new character index and calls PositionCaret essentially to convert that character index into a new pixel position of the caret.

Like TypeAway itself, this program unfortunately doesn't work correctly with text that is read (and typed) right to left.

**Scribbling with the Mouse**

You've heard of CAD programs? You've heard of paint programs? The Scribble program is neither of these.

**Scribble.cs**

//----------
// Scribble.cs © 2001 by Charles Petzold
//----------
using System;
using System.Drawing;
using System.Windows.Forms;

class Scribble: Form
{

bool bTracking;
Point ptLast;

public static void Main()
{
    Application.Run(new Scribble());
}

public Scribble()
{
    Text = "Scribble";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;
}

protected override void OnMouseDown(MouseEventArgs mea)
{
    if (mea.Button != MouseButtons.Left)
        return;

    ptLast = new Point(mea.X, mea.Y);
    bTracking = true;
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    if (!bTracking)
        return;

    Point ptNew = new Point(mea.X, mea.Y);

    Graphics grfx = CreateGraphics();
    grfx.DrawLine(new Pen(ForeColor), ptLast, ptNew);
    grfx.Dispose();
    ptLast = ptNew;
}

protected override void OnMouseUp(MouseEventArgs mea)
{
    bTracking = false;
}

protected override void OnPaint(PaintEventArgs pea)
{
    // What do I do here?
}

At first, the program seems to work just fine. You position the mouse cursor over the program’s client area, press the left mouse button, and drag the mouse to draw straight, curvy, or otherwise...
awkward-looking lines. (I'm using a simple approach to mouse tracking here so as not to overly obscure how the program works.) The drawing occurs during the `OnMouseMove` method: the program obtains a `Graphics` object from `CreateGraphics` and simply draws a line from the previous mouse position (which it has saved in the field `ptLast`) to the new mouse position. Here's my homage to the early advertisements for the Apple Macintosh:

```
```

But what does Scribble do during its `OnPaint` method? Oops! The program forgot to retain all those mouse positions it used to draw the lines. If the client area needs repainting, it's out of luck.

You can implement a repainting facility in a program such as this in a couple ways. One technique is to use a shadow bitmap that the program draws on at the same time it draws on the screen. During the `OnPaint` method, it simply displays that bitmap. I'll have another version of the Scribble program in Chapter 11 that does precisely this. In Chapter 15, I use a graphics path to save the points.

Another solution is to accumulate an array of `Point` structures and simply call `DrawLines` during the `OnPaint` method. Well, that raises some questions as well. The number of elements that a C# array can store is fixed at the time it's created. We might be tempted to create an array of many points, as in this example:
```
Point[] apt = new Point[1000000];
```

But we would burden ourselves with two conflicting fears: first, that we hadn't allocated enough points for a particular artistic user, and second, that we were wasting an awful lot of memory.

The solution is the `ArrayList` class, defined in the `System.Collections` namespace, which also includes classes with the mouthwatering names `Queue`, `Stack`, `SortedList`, and `Hashtable`. An `ArrayList` object is like a single-dimension array that expands itself when necessary. I can't go into a full discussion of `ArrayList` here, but I'll give you the basics and you can explore the rest on your own.

You begin by creating a new `ArrayList` object:
```
ArrayList arrlst = new ArrayList();
```

An alternative constructor provides an initial capacity. By default, the capacity is set to 16. Then you can use the `Add` method to add any object to `ArrayList`. Here's a statement adding a `Point` object:
```
arrlst.Add(pt);
```

You can also insert or remove items by using similar methods.

One convenient approach to retrieving objects from `ArrayList` is to use an indexer, much like an array. For example, if you know that the fourth item in `arrlst` is a `Point` structure, you can get it by using
```
Point pt = (Point) arrlst[3];
```
The cast is needed because the indexer returns an object of type Object.

You can add different types of objects to the same array list. For example, right after adding a Point to the array list you can add a Rectangle:

arrlst.Add(rect);

However, there may come a time when you want to copy the contents of an array list into a regular array (as I'll demonstrate shortly). A run-time error will be raised if you try to copy a Rectangle object into an array of Point structures.

The Capacity property of the array list indicates how many objects the array list is currently capable of holding. As you add objects to the array list (and perhaps remove some), the Count property indicates the number of objects in the array list. Count is always less than or equal to Capacity. If Count equals Capacity and you add another item, Capacity is doubled.

A version of Scribble that uses the ArrayList class to save all the Point structures can't make do with only one array list. A single ArrayList of Point structures would imply that all the points are connected with a single line. However, the user can press the left mouse button, scribble around, and release the mouse button multiple times. One ArrayList is needed to store these points as they're being drawn. But then that collection of points needs to be converted to an array of Point structures. Each array of Point structures needs to be stored in another ArrayList object.

In the ScribbleWithSave program, the main ArrayList object (the one storing Point arrays) is the field named arrlstApts. The arrlstPts field is used to store each collection of points as they're being drawn.

ScribbleWithSave.cs

using System;
using System.Collections;       // For ArrayList
using System.Drawing;
using System.Windows.Forms;

class ScribbleWithSave: Form
{
    ArrayList arrlstApts = new ArrayList();
    ArrayList arrlstPts;
    bool bTracking;

    public static void Main()
    {
        Application.Run(new ScribbleWithSave());
    }
    public ScribbleWithSave()
    {
        Text = "Scribble with Save";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (mea.Button != MouseButtons.Left)
return;

arrlstPts = new ArrayList();
arrlstPts.Add(new Point(mea.X, mea.Y));

bTracking = true;
}
protected override void OnMouseMove(MouseEventArgs mea)
{
    if (!bTracking)
        return;

    arrlstPts.Add(new Point(mea.X, mea.Y));

    Graphics grfx = CreateGraphics();
grfx.DrawLine(new Pen(ForeColor),
    (Point) arrlstPts[arrlstPts.Count - 2],
    (Point) arrlstPts[arrlstPts.Count - 1]);
grfx.Dispose();
}
protected override void OnMouseUp(MouseEventArgs mea)
{
    if (!bTracking)
        return;

    Point[] apt = (Point[]) arrlstPts.ToArray(typeof(Point));
    arrlstApts.Add(apt);
bTracking = false;
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Pen      pen  = new Pen(ForeColor);

    for (int i = 0; i < arrlstApts.Count; i++)
        grfx.DrawLines(pen, (Point[]) arrlstApts[i]);
}
}

The program creates a new ArrayList object whenever the user presses the left mouse button and
the program gets a call to OnMouseDown. The first member of the ArrayList object is the mouse
cursor position at that time:
arrlstPts.Add(new Point(mea.X, mea.Y));

The program adds additional members on each call to OnMouseMove.

On receipt of OnMouseUp, the program uses the ToArray method to convert the collection of Point
structures into a Point array:
Point[] apt = (Point[]) arrlstPts.ToArray(typeof(Point));

(There's an overload of the ToArray method that doesn't require an argument, but it returns an array of type Object. With the argument, the method returns an object of type Array.) That Point array is then added to arllstApts:

arrlstApts.Add(apt);

What's really nice is the OnPaint method. It simply loops through the elements of arllstApts and casts each one to the Point array that it passes to DrawLines. (I didn't think displaying the lines stored in arllstPts was quite as necessary, although code to do that could easily be added.)

Of course, ScribbleWithSave can't really save an indefinite number of points. At some point, it will run out of memory. To protect itself, the program should probably enclose the calls to the Add method of ArrayList in a try block. But I wasn't even quite sure how I would test such a thing, given that it would occur only after a great deal of drawing.
Chapter 9: Text and Fonts

Overview

In a graphical environment, the seemingly commonplace exercise of displaying text takes on additional layers of complexity. Fonts are often proportionally spaced, and they usually come in a variety of styles and sizes, which means that text must be handled much like other graphical output. Yet text is not quite like the abstract analytical geometry of lines and filled areas and thus occupies an uneasy niche in the field of computer graphics. To a typographer, fonts are a form of art with a long history of sophisticated design. Creators of computer graphics systems that implement fonts with any degree of integrity must deal with classical typographical concepts; application programmers must also learn about these concepts.

The most important principle is that text is meant to be read. There are subtleties in font design, font rendering, and page layout that affect readability even if the average person doesn't consciously notice them. Moreover, text is not purely content; the style in which the text is printed on the page can affect the interpretation of the content, either positively or adversely. You don't want a wedding invitation to look like an office memorandum, or a doctoral thesis to look like a magazine advertisement.

Fonts Under Windows

The 1992 introduction of Microsoft Windows 3.1 marked a major change in how Windows applications used fonts. Prior to that, most of the fonts available for the video display under Windows were bitmap fonts (also called raster fonts) stored in discrete sizes and generally not scalable to other sizes. Also available were stroke fonts (also called plotter or vector fonts) defined as polylines, but these were unattractive and rarely used.

Windows 3.1 introduced TrueType, which greatly enhanced the ability of programmers and users to work with text in a flexible manner. TrueType is an outline font technology developed by Apple and Microsoft and is supported by many font manufacturers. Outline fonts are continuously scalable and contain built-in hints that prevent distortions when the outlines are scaled to a particular pixel size and grid.

Outline fonts also lend themselves well to integration with other graphics operators. You've already seen in Chapter 7 how you can scale, rotate, and shear text. In Chapter 15, I'll demonstrate how you can make text output part of a graphics path and use that path for outlining, filling, or clipping. Chapter 19 is devoted to exercises I collectively call Font Fun.

In 1997, Adobe and Microsoft announced the OpenType font format, which combines TrueType and the Type 1 outline font format used in PostScript, Adobe's page-description language. (In the Fonts dialog box invokable from Control Panel, TrueType font files are associated with an icon containing the initials TT, and OpenType font files have an icon with an O.)

The U.S. edition of Windows 2000 comes with 42 TrueType and OpenType font files, and the CD-ROM includes another 83 font files for non-Latin alphabets. You can optionally get access to these additional fonts by installing additional language support from the Regional Options dialog box of Control Panel.

Although bitmap fonts and stroke fonts are still supported under Windows, they are not directly available to Windows Forms applications. A Windows Forms program has direct access to only TrueType and OpenType fonts. This is actually a good thing, for it means that Windows Forms applications can work consistently with all fonts to which they have access and can use them on both the video display and the printer.

Windows Forms supports anti-aliasing of TrueType and OpenType fonts, and it also supports ClearType, a technique announced by Microsoft in 1998 for taking advantage of the arrangement of color dots on LCD displays. I'll discuss font anti-aliasing and ClearType later in this chapter.

Talking Type

Typographers generally denote a particular font by a typeface name (often called simply a face name) and a point size (sometimes called the em size). Each typeface belongs to a type family. Type
families have simple names such as Bookman, Helvetica, Garamond, and Times. Each family often includes several variations:

- The strokes that make up the characters can be light or heavy in various degrees, described by typeface names such as Helvetica Ultra Light, Helvetica Thin, Helvetica Light, Helvetica Bold, Helvetica Heavy, and Helvetica Black.
- The widths of the individual characters can be narrower or wider than usual, for typeface names such as Helvetica Narrow, Helvetica Condensed, or Helvetica Extended.
- The characters can be slanted to the right, giving us typeface names such as Helvetica Italic or Helvetica Oblique. Strictly speaking, oblique refers to characters that are simply slanted, while italic implies that the characters are also stylistically somewhat different from the upright font. The appearance of the lowercase “a” is usually a good indication of whether a font is oblique (a) or italic (a).

These three variations can be combined in a single typeface name—for example, Helvetica Bold Extended Oblique. Typeface names can also include the name of the copyright holder of the font and perhaps a code number meaningful only to the font manufacturer.

When TrueType was first introduced in Windows, it was represented by 13 TrueType files (filename extension .ttf) associated with the following typeface names:

- Courier New
- Courier New Bold
- Courier New Italic
- Courier New Bold Italic
- Times New Roman
- Times New Roman Bold
- Times New Roman Italic
- Times New Roman Bold Italic
- Arial
- Arial Bold
- Arial Italic
- Arial Bold Italic
- Symbol

Courier is a fixed-pitch font family that resembles typewriter output. Very little text is displayed in Courier these days, the big exceptions being command-line windows, program listings, and hex dumps.

Times New Roman is a clone of the Times font (renamed for copyright reasons) originally designed for the *Times of London* and used in much printed material. It is considered highly readable. Arial is a clone of Helvetica, a popular sans serif font. Serifs are small turns that often finish the strokes of letters. A sans serif font doesn't have serifs. (A font with serifs is sometimes called a roman font.) The Symbol font includes common symbols rather than letters.

Windows 2000 also includes additional TrueType and OpenType font files installed with the North American English version. These include the following eight roman font faces:

- Georgia
- Georgia Bold
- Georgia Italic
- Georgia Bold Italic
- Palatino Linotype
- Palatino Linotype Bold
- Palatino Linotype Italic
- Palatino Linotype Bold Italic

Fourteen sans serif font faces:

- Arial Black
- Impact
- Lucida Sans Unicode
- Microsoft Sans Serif
- Tahoma
- Tahoma Bold
- Trebuchet MS
- Trebuchet MS Bold
- Trebuchet MS Italic
A whimsical font face:
- Comic Sans MS
- Comic Sans MS Bold

Another fixed-pitch font face:
- Lucida Console

And two additional symbol font faces:
- Wingdings
- Webdings

As you can see, many (but not all) of the font families come in regular, bold, italic, and bold italic faces. In addition, Windows can apply underlining or strikeout to any font.

In graphical environments, users tend to use the word font to refer to what is technically a font family. "Let's change this font from Helvetica to Verdana," a user will say. In addition, users tend to think of italic and boldface (as well as underlining and strikeout) as attributes or styles that are applied to a particular font. For example, no user says, "I want to make this word italic so I have to switch the typeface name from Linotype Palatino to Linotype Palatino Italic." No, it's more like "I want to make this word italic," regardless of the face name.

Windows Forms helps you present fonts to the user in a manner familiar to the user's expectations by consolidating multiple face names (such as Arial, Arial Bold, Arial Italic, and Arial Bold Italic) into a single font family (Arial). Despite the theoretical wide variety of face names possible with different levels of stroke width and character width, the only variations allowed are a combination of bold, italic, underline, and strikeout styles. In Windows Forms, the Arial Black typeface is not considered part of the Arial family; Arial Black is considered a separate font family, which just so happens is not available on the Windows 2000 CD-ROM in italic or bold versions.

Font Heights and Line Spacing

Along with a typeface name, a font is identified by a vertical size in points. In traditional typography, a point is 0.01384 inch. This number is very close to 1/72 of an inch, so in computer typography, the point is assumed to be exactly 1/72 of an inch.

The point size of a font is commonly described as the height of the characters in the Latin alphabet—that is, the uppercase and lowercase letters A through Z without diacritical marks—from the very top of the ascenders to the bottom of the descenders, encompassing, for example, the full height of the characters "bq." That's certainly a convenient way to think of the point size, but it's usually not metrically precise.

Back in the days of metal type, the point size of a font was the vertical size of the metal type on which the letters were cast. The letters themselves were generally a little shorter than the point size. Today, this restriction has disappeared, and it's sometimes the case that letters can be larger than the point size. It's safer to think of the point size of a font as a typographical design concept rather than a metrical concept. The size of the characters in a particular font could be greater than or less than what the point size implies. Never assume that the point size of a font is anything other than an approximate measure of the height of the font characters.

Getting familiar with common point sizes is helpful when beginning to work with fonts. Most of The New York Times is printed in 8-point type; Newsweek is 9-point type; this book has 10.5-point type. The default Windows font is 10 points. The default Windows Forms font is about 8 points. As I discussed in Chapter 7, the user is responsible for setting an assumed resolution of the video display, and that resolution is what affects the visual size of these 8-point and 10-point fonts.

I mentioned earlier that the point size is sometimes referred to as the em size. The term comes from the size of the square piece of metal type used in olden days for the capital M. These days, the em is used mostly to refer to horizontal measurements. The width of an em in a particular font is equal to the vertical point size of the font. For example, in a 14-point font, the em dash and the em space are
both 14 points wide. The en is half of the em. In a 14-point font, the en dash and the en space are 7 points wide.

Successive lines of text are generally spaced by an amount somewhat larger than the point size, usually at least about 115 percent of the point size. The rationale for the line spacing is partly based on the need for some extra space for the diacritics that appear in many European languages. But line spacing is also an aesthetic necessity: text is easier to read if there's some air between the lines.

The recommended line spacing is the value you obtain from the `Height` property and the `GetHeight` method of the `Font` class (both of which I'll discuss later in this chapter in context with other `Font` properties). For many fonts, the recommended line spacing is usually larger than the point size but somewhat smaller than the height returned from the `MeasureString` method of the `Graphics` class. As I've mentioned before, you should avoid using the `Height` property unless you know that you're dealing only with the default page transform on the video display. Because `Height` doesn't involve a `Graphics` object, it isn't applicable for the printer or for nondefault page transforms.

**Default Fonts**

Since Chapter 2, we've been using the `Font` property that's implemented in `Control` and inherited by all its descendents, including `Form`:

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font</td>
<td>Font</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

As you'll see shortly, you can set this `Font` property to a different `Font` object and all the successive `DrawString` calls that use the `Font` property will use this different font. (You can also just create a new `Font` object and use that directly in `DrawString`.) If you change the value of the `Font` property, you can set it back to the original value by using this static read-only property of `Control`:

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font</td>
<td>DefaultFont</td>
<td>get</td>
</tr>
</tbody>
</table>
```

You can do this manually:

```csharp
Font = DefaultFont;
```

Or you can simply use this method:

```csharp
Control Methods (selection)
void ResetFont();
```

There's also this font-related property:

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>FontHeight</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

You can use this property instead of `Font.Height`. Although this property is writable, setting it to a new value won't change the `Font` property!

**Variation on a Font**

The `System.Drawing` namespace defines two important classes for working with fonts:

- `FontFamily` is identified by a string such as "Times New Roman."
- `Font` is a combination of a font family (either a `FontFamily` object or a character string identifying the family name), attributes (such as italic and bold), and a point size.
I’m going to begin with the Font class. The Font constructors are in three categories:

- Based on an existing Font object
- Based on a character string identifying the font family
- Based on a FontFamily object

The simplest constructor for Font creates a new font based on an existing font. The new font is the same except for the font style:

**Font Constructors (selection)**

Font(Font font, FontStyle fs)

*FontStyle* is an enumeration defined as a series of single-bit flags:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>0</td>
</tr>
<tr>
<td>Bold</td>
<td>1</td>
</tr>
<tr>
<td>Italic</td>
<td>2</td>
</tr>
<tr>
<td>Underline</td>
<td>4</td>
</tr>
<tr>
<td>Strikeout</td>
<td>8</td>
</tr>
</tbody>
</table>

For example, suppose *font* is an existing font, perhaps obtained from the form’s Font property:

```csharp
Font font = Font;
```

You can make an italic version of that font named *fontItalic* by using the following statement:

```csharp
Font fontItalic = new Font(font, FontStyle.Italic);
```

You can use multiple enumeration members combined with the C# bitwise OR operator:

```csharp
Font fontBoldStrikeout = new Font(font, FontStyle.Bold | FontStyle.Strikeout);
```

Here’s a program that takes the form’s Font property and creates bold and italic versions for displaying a mix of regular, bold, and italic text.

**BoldAndItalic.cs**

```csharp
// BoldAndItalic.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class BoldAndItalic : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new BoldAndItalic());
    }

    public BoldAndItalic()
    {
```
isplay override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
  const string str1 = "This is some ";
  const string str2 = "bold";
  const string str3 = " text and this is some ";
  const string str4 = "italic";
  const string str5 = " text.";
  Brush brush = new SolidBrush(clr);
  Font fontRegular = Font;
  Font fontBold = new Font(fontRegular, FontStyle.Bold);
  Font fontItalic = new Font(fontRegular, FontStyle.Italic);
  float x = 0;
  float y = 0;

  grfx.DrawString(str1, fontRegular, brush, x, y);
  x += grfx.MeasureString(str1, fontRegular).Width;

  grfx.DrawString(str2, fontBold, brush, x, y);
  x += grfx.MeasureString(str2, fontBold).Width;

  grfx.DrawString(str3, fontRegular, brush, x, y);
  x += grfx.MeasureString(str3, fontRegular).Width;

  grfx.DrawString(str4, fontItalic, brush, x, y);
  x += grfx.MeasureString(str4, fontItalic).Width;

  grfx.DrawString(str5, fontRegular, brush, x, y);
}

Because DrawString has a Font argument and a particular font is either regular, bold, italic, or bold italic, multiple DrawString calls are required to display text that combines multiple styles. The program uses MeasureString to determine the size of each piece of text and space the text horizontally.

If you look closely, you'll probably notice that there seems to be a little extraneous space between each piece of displayed text. I'll show you how to avoid this extra space when I get to detailed coverage of the StringFormat class later in this chapter.

Let me give you another warning now, and we'll examine later how to deal with it: this Font constructor can fail if the particular font family that the font belongs to isn't capable of the requested style. The constructor throws an exception that a well-behaved program should deal with. This isn't a
problem in the BoldAndItalic program because it's using the form's default font, and that font is capable of all the styles. The BoldAndItalic program will not work with every font family, however.

**Creating Fonts by Name**

This next set of constructors for *Font* is exceptionally convenient and straightforward. You specify a font by using the font family name, the point size, and an optional style:

```
Font Constructors (selection)
```

```
Font(string strFamily, float fSizeInPoints)
Font(string strFamily, float fSizeInPoints, FontStyle fs)
```

The font family names you can use as the first argument to the *Font* constructor are familiar names such as "Times New Roman," "Arial," "Courier New," "Comic Sans MS," and many others. For example,

```
Font font = new Font("Times New Roman", 24);
```

creates a 24-point Times New Roman font.

I love creating fonts like this, and I suspect that you too will find yourself using these *Font* constructors more than the others. However, you should keep in mind some drawbacks.

The name should represent a TrueType or OpenType font that is installed on the system on which the program is running. If the Times New Roman font isn't available—or if you misspell the name—the constructor will substitute a default font (probably Microsoft Sans Serif). Can you be sure that the Times New Roman font is available? Well, yes, if you're coding something for yourself. And yes again, if you're coding something for internal use in a company where you're sure that all the machines have Times New Roman fonts installed. But it's possible for users to uninstall TrueType fonts, and while getting rid of Times New Roman may sound perverse to you and me, it's not impossible. As your code achieves an ever broader platform base, using explicit font family names becomes less safe. At some point in the future, Windows Forms programs might run under environments that have other collections of fonts with different names. Presumably, those environments will implement some kind of font-mapping so that existing programs don't break, but it's probably still risky to use the more obscure font family names.

Sticking to the familiar three font family names of Times New Roman, Arial, and Courier New is probably safest. Some aliases are even allowed: you can use "Times" for Times New Roman and "Helvetica" for Arial.

Specifying an explicit point size is less problematic. You know that the user has set the video display properties based on the idea that a 10-point font is comfortable. Windows Forms itself sets the form's *Font* property based on the assumption that an 8-point font is also readable. Everything else is relative. For example, a 24-point font is three times larger than the normal Windows Forms font.

Because there are 72 points to the inch, a 24-point font is approximately 1/3 inch tall. (I say *approximately* because, as I mentioned earlier, the point size is a typographical design concept, not a precise measurement.) You can also think of a 24-point font as having a size in pixels that is approximately 1/3 the *DpiY* property of the *Graphics* object.

The family name and the point size can also be combined with a font style. The following program creates and displays 18-point Courier New, Arial, and Times New Roman fonts in regular, bold, italic, and bold-italic versions. These 18-point fonts are approximately 1/4 inch in size.

```
FontNames.cs
```

```
using System;
using System.Drawing;
```

using System.Windows.Forms;

class FontNames: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FontNames());
    }
    public FontNames()
    {
        Text = "Font Names";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        string[] astrFonts = { "Courier New", "Arial", "Times New Roman" };
        Brush brush = new SolidBrush(clr);
        float y = 0;

        foreach (string strFont in astrFonts)
        {
            foreach (FontStyle fs in afs)
            {
                Font font = new Font(strFont, 18, fs);
                grfx.DrawString(strFont, font, brush, 0, y);
                y += font.GetHeight(grfx);
            }
        }
    }
}

This class derives from PrintableForm, so you can print the fonts by clicking on the client area. Keep in mind that the coordinates passed to the DrawString method indicate the position of the upper left corner of the string. Therefore, coordinates for each string must be adjusted by the text height of the previous string. The program adjusts the coordinate by using the Font method GetHeight after displaying text using the font.

Notice also that the program assumes that each font returns a different value from the GetHeight method. Put a Console.WriteLine statement in there if you're curious about these values. You'll find that the Times New Roman and Arial fonts return the same value, which is a little larger than the Courier New value. But other fonts may be quite different, and there's no reason you should guess. Use GetHeight to make sure. Here's the program's display:
You can try substituting the Height property for the GetHeight method and see what happens when you print the output on your printer. The line spacing will be off by an amount that's dependent on how much the video display resolution in dots per inch (dpi) differs from the 100-dpi resolution set for the printer.

Here's a very similar program that displays the Times New Roman font in sizes from 6 points to 12 points in increments of 1/4 point.

FontSizes.cs

//------------------------
// FontSizes.cs © 2001 by Charles Petzold
//------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FontSizes: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FontSizes());
    }
    public FontSizes()
    {
        Text = "Font Sizes";
    }
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    string strFont = "Times New Roman";
    Brush brush = new SolidBrush(clr);
    float y = 0;

    for (float fSize = 6; fSize <= 12; fSize += 0.25f)
    {
        Font font = new Font(strFont, fSize);
        grfx.DrawString(strFont + " in " + fSize + " points",
                        font, brush, 0, y);
        y += font.GetHeight(grfx);
    }
}

Here's the program output:

![Font Sizes](image)

Perhaps what's most noticeable about this display is the sudden leap at 10.5 points from strokes that are 1 pixel wide to strokes that are 2 pixels wide. Such transitions are not evident on higher-resolution devices such as printers.

If you want to use a larger font for everything your program displays in its client area, you can change the Font property of a form right in its constructor. Here's a program that overrides the BoldAndItalic program shown earlier and displays the text string with a 24-point font.
BoldAndItalicBigger.cs
//--------------------------------------------------
// BoldAndItalicBigger.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class BoldAndItalicBigger: BoldAndItalic
{
    public new static void Main()
    {
        Application.Run(new BoldAndItalicBigger());
    }
    public BoldAndItalicBigger()
    {
        Text += " Bigger";
        Font = new Font("Times New Roman", 24);
    }
}

Here's the program display:

This is some **bold** text and this is some *italic* text.

In this program, the extraneous space between the various pieces of text has become more
noticeable than in the BoldAndItalic program. Avoiding this extra space requires a StringFormat
object, as I'll demonstrate later in this chapter.

**A Point Size by Any Other Name...**

You need not specify the size of the font in points. Two more constructors for Font include a
GraphicsUnit argument:

**Font Constructors (selection)**

Font(string strFamily, float fSize, GraphicsUnit gu)
Font(string strFamily, float fSize, FontStyle fs, GraphicsUnit gu)

You can use all but one of the GraphicsUnit enumeration values that you learned about in
connection with the PageUnit property in Chapter 7:

**GraphicsUnit Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>0</td>
<td>Units of world coordinates</td>
</tr>
<tr>
<td>Display</td>
<td>1</td>
<td>Won't work with Font constructor!</td>
</tr>
<tr>
<td>Pixel</td>
<td>2</td>
<td>Units of pixels</td>
</tr>
<tr>
<td>Point</td>
<td>3</td>
<td>Units of 1/72 inch</td>
</tr>
</tbody>
</table>
### GraphicsUnit Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inch</td>
<td>4</td>
<td>Units of inches</td>
</tr>
<tr>
<td>Document</td>
<td>5</td>
<td>Units of 1/300 inch</td>
</tr>
<tr>
<td>Millimeter</td>
<td>6</td>
<td>Units of millimeters</td>
</tr>
</tbody>
</table>

The constructor

```csharp
new Font(strFamily, float fSize)
```

is identical to

```csharp
new Font(strFamily, float fSize, GraphicsUnit.Point)
```

Indeed, the following constructors are all equivalent:

```csharp
new Font(strFamily, 72)
new Font(strFamily, 72, GraphicsUnit.Point)
new Font(strFamily, 1, GraphicsUnit.Inch)
new Font(strFamily, 25.4f, GraphicsUnit.Millimeter)
new Font(strFamily, 300, GraphicsUnit.Document)
```

All these constructors result in the creation of identical 72-point fonts. There's nothing going on here that's more sophisticated than knowing that an inch is equal to 72 points and 25.4 millimeters.

The messy `Font` constructor arguments are `GraphicsUnit.Pixel` and `GraphicsUnit.World`. For the video display, if you're displaying text with the default page transform (that is, all coordinates and sizes are in units of pixels), you can also use the following two constructors to create 72-point fonts:

```csharp
new Font(strFamily, grfx.DpiY, GraphicsUnit.Pixel)
new Font(strFamily, grfx.DpiY, GraphicsUnit.World)
```

The second argument is the number of pixels in one vertical inch.

The equivalence of these constructors is demonstrated in the following program, which creates 24-point fonts seven different ways.

### TwentyFourPointScreenFonts.cs

```csharp
//---------------------------------------------------------
// TwentyFourPointScreenFonts.cs © 2001 by Charles Petzold
//---------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwentyFourPointScreenFonts : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TwentyFourPointScreenFonts());
    }

    public TwentyFourPointScreenFonts()
    {
        Text = "Twenty-Four Point Screen Fonts";
    }
}
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Brush brush = new SolidBrush(clr);
    float y = 0;
    Font font;
    string strFamily = "Times New Roman";
    
    font = new Font(strFamily, 24);
    grfx.DrawString("No GraphicsUnit, 24 points", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, 24, GraphicsUnit.Point);
    grfx.DrawString("GraphicsUnit.Point, 24 units", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, 1 / 3f, GraphicsUnit.Inch);
    grfx.DrawString("GraphicsUnit.Inch, 1/3 units", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, 25.4f / 3, GraphicsUnit.Millimeter);
    grfx.DrawString("GraphicsUnit.Millimeter, 25.4/3 units", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, 100, GraphicsUnit.Document);
    grfx.DrawString("GraphicsUnit.Document, 100 units", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, grfx.DpiY / 3, GraphicsUnit.Pixel);
    grfx.DrawString("GraphicsUnit.Pixel, " + grfx.DpiY / 3 + " units", font, brush, 0, y);
    y += font.GetHeight(grfx);
    
    font = new Font(strFamily, grfx.DpiY / 3, GraphicsUnit.World);
    grfx.DrawString("GraphicsUnit.World, " + grfx.DpiY / 3 + " units", font, brush, 0, y);
}
I'm using 24-point fonts rather than 72-point fonts in this program just so they all fit on the display. In each of the constructors, the values passed as the second argument are simply 1/3 the values I showed previously.

On the video display, all seven lines of text are the same height. If you click on the client area of this program to print the output, however, you'll discover a problem. The first five lines of output look fine. These constructors have all successfully created 24-point fonts for the printer. But the last two lines create fonts that are much too large.

As you'll recall, the DpiX and DpiY properties of the Graphics object for the printer give its true resolution: probably 300, 600, 720, or something higher. In the final two Font constructors, the program specifies 1/3 that resolution, so the second argument will be 100, 200, 240, or something higher. The default page transform for the printer makes it appear to be a 100-dpi device. The combination of the font size and page transform results in a font that is 1 inch, 2 inches, 2.3 inches, or something larger.

To create 72-point fonts for the printer's default page transform using GraphicsUnit.Pixel or GraphicsUnit.World, you need to use the following constructors:

```csharp
new Font(strFamily, 100, GraphicsUnit.Pixel)
new Font(strFamily, 100, GraphicsUnit.World)
```

To create 24-point fonts for the printer, you need to use 1/3 of 100. The following program is the same as the previous one except that the last two Font constructors create 24-point fonts appropriate for the printer.

```csharp
public TwentyFourPointPrinterFonts() {
    Text = "Twenty-Four Point Printer Fonts";
}
```
On the printer, all seven lines of text will be the same height. On the video display, the first five lines will be 24-point fonts, and the last two will probably be a little off, depending on how much your video display resolution differs from 100 dpi.

**Clash of the Units**

Experimenting with different units in the `Font` constructors raises the question, How do the font units interact with the world transform and the page transform? Both the `Font` class and the `Graphics` class make use of the `GraphicsUnit` enumeration. The `Font` class uses the `GraphicsUnit` enumeration in some of its constructors, and the `Graphics` class `PageUnit` property is also set equal to one of the enumeration values.

We've already had a little taste of that interaction. Now let's see if we can come up with an overall analysis and a few solid rules.

Try to keep in mind that `Font` objects are device-independent. It doesn't matter what world transform or page transform is in effect when you create the font. The `Font` constructor doesn't know anything
about that. You can create Font objects anywhere in your program regardless of whether or not there's a Graphics object in sight.

There are only three commonly used methods that involve the interaction of both a Font object (which is created in a particular size with particular units) and a Graphics object (which has a world transform and a page transform associated with it):

§ DrawString, a method of the Graphics class that has a Font argument
§ MeasureString, a method of the Graphics class that has a Font argument
§ GetHeight, a method of the Font class that has a Graphics argument

These are generally the only three methods in which you have to worry about the clash among graphics units and transforms. Two others—the DrawStringDisabled method of the ControlPaint class and the MeasureCharacterRanges method of the Graphics class—aren't used nearly as often.

Only three rules affect these methods. I encourage you to experiment with the TwentyFourPointScreenFonts and TwentyFourPointPrinterFonts programs to verify that what I say is correct:

Rule 1  The world transform affects everything in the same way.

Let's say you have some graphics output: a collection of lines, filled areas, and text using fonts created with a variety of GraphicsUnit arguments. Then you decide you want everything twice as big. So, before any of the graphics output calls, you put the statement

grfx.ScaleTransform(2, 2);

The world transform affects everything in the same way. Everything—every line, every filled area, and every text string—is doubled in size regardless of the way in which the font was created. The sizes returned from MeasureString and GetHeight remain the same, however.

Rule 2  For fonts constructed with metrical sizes (that is, units of points, inches, or millimeters), the page transform doesn't affect the physical size of the text.

Let's say you create a 72-point font:

Font font = new Font("Arial", 72, GraphicsUnit.Point);

And you also decide you'd like to draw in units of millimeters:

grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;

Regardless of the page transform, the physical size of the text remains the same. Because the font size is 72 points, it's equivalent to a height of about 25.4 units, where the units are millimeters.

What the page transform does affect, however, are the coordinates to the DrawString method, the sizes returned from the MeasureString method, and the height returned by GetHeight. All those coordinates and sizes are in units of millimeters. This should be OK, however, because you're generally using GetHeight or MeasureString to calculate the coordinates you pass to DrawString. Just make sure that the same transforms are in effect when you obtain numbers from GetHeight or MeasureString as when you pass the numbers to DrawString.

If you insert statements into the TwentyFourPointScreenFonts program to change the PageUnit and PageScale properties, you'll find that the first five lines of text are unaffected. The last lines are affected by the page transform, however. That's the third rule.

Rule 3  For fonts constructed with GraphicsUnit.Pixel or GraphicsUnit.World units, the size of the font is assumed to be in units of world coordinates.

In other words, the font size is treated just like the coordinates and sizes passed to the various line-drawing and area-filling methods of the Graphics class. For example, suppose you create a font like so:

Font font = new Font("Arial", 72, GraphicsUnit.World);
With the default page transform, that's assumed to be 72 pixels on the video display or 72/100 inch on the printer. If you set the page transform to millimeters, as here,

```csharp
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
```

the font size of 72 is assumed to be units of millimeters, which results in fonts almost 3 inches in height.

For fonts created with `GraphicsUnit.World` and `GraphicsUnit.Pixel`, the values returned from `GetHeight` and `MeasureString` are unaffected by the page transform. The physical size of the font is in units indicated by the page transform, and the sizes returned from these methods are also in page units.

Finally, although their names would imply otherwise, I have discovered no difference between the `GraphicsUnit.Pixel` and `GraphicsUnit.World` arguments when used to create fonts.

**Font Properties and Methods**

All properties of the `Font` class are read-only. That implies that you can't make a font a little different simply by changing one of its properties. Here's the complete list of `Font` properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Name</td>
<td>get</td>
<td>Font family name</td>
</tr>
<tr>
<td><code>FontFamily</code></td>
<td><code>FontFamily</code></td>
<td>get</td>
<td>Font family class</td>
</tr>
<tr>
<td><code>FontStyle</code></td>
<td><code>Style</code></td>
<td>get</td>
<td>From constructor</td>
</tr>
<tr>
<td>bool</td>
<td><code>Bold</code></td>
<td>get</td>
<td>True if boldface</td>
</tr>
<tr>
<td>bool</td>
<td><code>Italic</code></td>
<td>get</td>
<td>True if italic</td>
</tr>
<tr>
<td>bool</td>
<td><code>Underline</code></td>
<td>get</td>
<td>True if underlined</td>
</tr>
<tr>
<td>bool</td>
<td><code>Strikeout</code></td>
<td>get</td>
<td>True if strikeout</td>
</tr>
<tr>
<td>float</td>
<td><code>Size</code></td>
<td>get</td>
<td>From constructor</td>
</tr>
<tr>
<td><code>GraphicsUnit</code></td>
<td><code>Unit</code></td>
<td>get</td>
<td>From constructor</td>
</tr>
<tr>
<td>float</td>
<td><code>SizeInPoints</code></td>
<td>get</td>
<td>Calculated from <code>Size</code></td>
</tr>
<tr>
<td>int</td>
<td><code>Height</code></td>
<td>get</td>
<td>Line spacing for video display</td>
</tr>
<tr>
<td>byte</td>
<td><code>GdiCharSet</code></td>
<td>get</td>
<td>GDI character set ID</td>
</tr>
<tr>
<td>bool</td>
<td><code>GdiVerticalFont</code></td>
<td>get</td>
<td>True if a vertical font</td>
</tr>
</tbody>
</table>

The `Size` and `Unit` properties just return the values used to create the font. The `SizeInPoints` property is calculated from these values. For `GraphicsUnit.Pixel` and `GraphicsUnit.World`, the calculation is based on the resolution of the video display.

If you don't need to interface with Win32 API code, there's really only one `Font` method that's of any interest, and that's one I've already emphasized. It comes in three different versions:

**Font Methods (selection)**

```csharp
float GetHeight()
float GetHeight(Graphics grfx)
float GetHeight(float fDpi)
```
The value returned from `GetHeight` is what you should use for spacing successive lines of text. The version without an argument applies only to the video display in its default page transform. The second version is the most useful and takes the resolution and page transform of the output device into account. The third version obtains the line spacing based on a hypothetical vertical resolution in dots per inch.

If you need to interface with Win32 API code, `Font` has three static methods that you can use to create a `Font` object: `FromHdc`, `FromHfont`, and `FromLogFont`. Otherwise, the `Font` class doesn't have any way to create a `Font` object other than to use one of the constructors of the class. I've already discussed five of these constructors; four more are coming up soon. As you'll see in Chapter 16, the `FontDialog` class displays a dialog box that lets the user select a font and creates a `Font` object that applications can use.

Here's a program that displays all the properties of the form's `Font` property as well as the result of three versions of `GetHeight`, the third using a resolution of 100 dpi.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class AllAboutFont : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new AllAboutFont());
    }

    public AllAboutFont()
    {
        Text = "All About Font";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawString(
            "Name: " + Font.Name + "\n" +
            "FontFamily: " + Font.FontFamily + "\n" +
            "FontStyle: " + Font.Style + "\n" +
            "Bold: " + Font.Bold + "\n" +
            "Italic: " + Font.Italic + "\n" +
            "Underline: " + Font.Underline + "\n" +
            "Strikeout: " + Font.Strikeout + "\n" +
            "Size: " + Font.Size + "\n" +
            "GraphicsUnit: " + Font.Unit + "\n" +
            "SizeInPoints: " + Font.SizeInPoints + "\n" +
            "Height: " + Font.Height + "\n" +
            "GdiCharSet: " + Font.GdiCharSet + "\n" +
            "GdiVerticalFont: " + Font.GdiVerticalFont + "\n" +
            "GetHeight(0): " + GetHeight(0) + "\n" +
            "GetHeight(100): " + GetHeight(100) + "\n" +
            "GetHeight(200): " + GetHeight(200) + "\n" +
        );
    }
}
```
Try clicking on this program's client area to print this output. The printed output will be the same as
the display output except for the penultimate GetHeight value, which on the printer will match the last
value. You can look at the properties of other fonts by simply setting the form's Font property to the
font you want to examine. Here's what the program looks like on my system:

I have my video display settings set for Large Fonts, which implies a resolution of 120 dpi. The line-
spacing value of 14.71 pixels corresponds to about 0.123 inch, or about 9 points, a suitable line
spacing for an 8-point font. If you have Small Fonts installed (for an implied resolution of 96 dpi), the
line-spacing value will be 12.45; the Height property returns 13.

The first constructor for Font that we looked at creates a new font based on an existing font but with
a different style property. There are times when you need to do something similar but with a different
size.

For example, suppose you want to create a font that fits the interior of a specified rectangle. To do
this, you need to start off with a font and the text you want to display. You use MeasureString to
determine the dimensions of the displayed string, and then you create a new font with a size that's
scaled to the size of the rectangle. Here's a program that displays the text "Howdy, world!" (a
variation on the traditional text to include a character with a descender). The text is scaled as large
as possible to fit in the client area.

HowdyWorld.cs
//-----------------------------------------
// HowdyWorld.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HowdyWorld: PrintableForm
{
public new static void Main()
{
    Application.Run(new HowdyWorld());
}
public HowdyWorld()
{
    Text = "Howdy, world!";
    MinimumSize = SystemInformation.MinimumWindowSize + new Size(0,1);
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Font font = new Font("Times New Roman", 10, FontStyle.Italic);
    SizeF sizef = grfx.MeasureString(Text, font);
    float fScale = Math.Min(cx / sizef.Width, cy / sizef.Height);

    font = new Font(font.Name, fScale * font.SizeInPoints, font.Style);
    sizef = grfx.MeasureString(Text, font);

    grfx.DrawString(Text, font, new SolidBrush(clr),
    (cx - sizef.Width) / 2, (cy - sizef.Height) / 2);
}

The setting of the MinimumSize property in the constructor prevents the client area height from going to zero, which would result in a zero font size and an exception being thrown.

The DoPage method begins with the creation of a Font object, but the font this method uses could just as well have been the form's Font property or a font created somewhere else. The idea is that the program doesn't really need to know what arguments were originally used to create the font.

The next statement in DoPage uses MeasureString to find the length of a string (which happens to be the form's Text property) based on the font we just created, and the third statement calculates a scaling factor based on the relationship between the size of the client area (or the printable area of the printer page) and the size of the text. Notice the use of the Math.Min method to find the minimum of the horizontal and vertical scaling factors.

Next, the DoPage method creates a new font based on the existing font but scaling the point size by the fScale factor. MeasureString is called again. (Alternatively, you can multiply the Width and Height of the previous SizeF object by fScale.) Finally, the method centers the string in its client area.
If the window is extra wide, the size of the font is governed by the window height:

![Howdy, world!](image)

Is it possible to create a font that fills up the rectangle regardless of its aspect ratio? Of course, the characters would be distorted—either wider or narrower than the font height would imply. You can't do something like this with the `Font` constructors. In the `Font` constructors, you specify a font height. The character widths are based on that height.

However, you can distort the aspect ratio of text characters by using the world transform. Here's a variation of the HowdyWorld program called HowdyWorldFullFit that does just that.

**HowdyWorldFullFit.cs**

```csharp
//------------------------------------------------
// HowdyWorldFullFit.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HowdyWorldFullFit: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new HowdyWorldFullFit());
    }
    public HowdyWorldFullFit()
    {
        Text = "Howdy, world!";
        MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Font font = new Font("Times New Roman", 10, FontStyle.Italic);
        SizeF sizef = grfx.MeasureString(Text, font);
        float fScaleHorz = cx / sizef.Width;
        float fScaleVert = cy / sizef.Height;

        grfx.ScaleTransform(fScaleHorz, fScaleVert);

        grfx.DrawString(Text, font, new SolidBrush(clr), 0, 0);
    }
}
```
This version calculates separate scaling factors for the horizontal and vertical dimensions. The scaling factors are passed directly to the `ScaleTransform` method. The `DrawString` call displays the text at point (0, 0):

![Image of text: Howdy, world!]

As usual, you can print this out and see the characters extend to fit almost the height of the paper.

Notice in this screen shot (and the second screen shot from the first HowdyWorld program) that the text doesn't extend to the full height of the client area. There's ample space above the ascenders that isn't being used for anything. That space is mostly used for diacritical marks. If the text contained characters À, Å, or Ä (Unicode characters \x00C0, \x00C1, and \x00C2), for example, the accent marks would reach to the top of the client area.

Using the Windows Forms libraries, it isn't possible for an application to determine the amount of space that is reserved for diacritical marks. Nor is it possible to directly determine the x-height of a font, which is the height of lowercase letters without ascenders (such as "x") above the baseline. These particular font metrics, as they're called, are not exposed in the Windows Forms libraries. But other font metrics are derivable, as I'll discuss later in this chapter, and if you really need the information, you can approximately determine x-heights using paths. (Paths are covered in Chapter 15.)

**New Fonts from `FontFamily`**

Four more constructors for `Font` are the same as the previous four constructors except that the first argument is a `FontFamily` object rather than a string with the family name:

**Font Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Font(FontFamily ff, float fSizeInPoints)</code></td>
</tr>
<tr>
<td><code>Font(FontFamily ff, float fSizeInPoints, FontStyle fs)</code></td>
</tr>
<tr>
<td><code>Font(FontFamily ff, float fSize, GraphicsUnit gu)</code></td>
</tr>
<tr>
<td><code>Font(FontFamily ff, float fSize, FontStyle fs, GraphicsUnit gu)</code></td>
</tr>
</tbody>
</table>

So the question now becomes, How do you get a `FontFamily` object?

One way you can get it is from an existing font. For example, if you wanted to create a new font based on an existing font but in a new size, you could reference the `Name` property of the existing font as I did in the HowdyWorld program:

```csharp
Font font18 = new Font(font.Name, 18, font.Style);
```
Or you could use the `FontFamily` property as the first argument:

```csharp
Font font18 = new Font(font.FontFamily, 18, font.Style);
```

Another way to get a `FontFamily` object is to use one of the three `FontFamily` constructors:

### `FontFamily` Constructors

- `FontFamily(string strFamily)`
- `FontFamily(GenericFontFamilies gff)`
- `FontFamily(string strFamily, FontCollection fontcoll)`

That first constructor strongly suggests that a `FontFamily` object is defined entirely by a font family name. Indeed, the familiar statement

```csharp
Font font = new Font(strFamily, fSizeInPoints);
```

is just a shortcut for

```csharp
Font font = new Font(new FontFamily(strFamily), fSizeInPoints);
```

The only nonstatic property of `FontFamily` is its name:

### `FontFamily` Nonstatic Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Name</td>
<td>get</td>
<td><code>FontFamily</code> name</td>
</tr>
</tbody>
</table>

Although you’ve seen how you can create a `Font` without explicitly creating a `FontFamily`, it's sometimes useful to get the `FontFamily` first and store that in its own variable:

```csharp
FontFamily ff = new FontFamily(strFamily);
```

You can then use the `IsStyleAvailable` method to determine whether a particular style is available:

### `FontFamily` Methods (selection)

```csharp
bool IsStyleAvailable(FontStyle fs)
```

Not all TrueType or OpenType fonts have bold or italic versions, and if you try to create an italic or a bold font with a style that's not supported, you'll generate an exception. Worse yet, not all fonts have regular versions! It makes more sense to have code like this:

```csharp
if (ff.IsStyleAvailable(FontStyle.Italic))
    fontItalic = new Font(ff, 24, FontStyle.Italic);
else if (ff.IsStyleAvailable(FontStyle.Regular))
    fontItalic = new Font(ff, 24, FontStyle.Regular);
else
    fontItalic = new Font(ff, 24, FontStyle.Bold);
```

This code might not result in creating an italic font, but at least you've avoided raising the exception.

As you saw at the beginning of this chapter, for many font families, separate files support the italic, bold, and bold italic versions of the families. In some cases, Windows synthesizes italic and bold, which means it creates italic, bold, and bold italic versions by modifying the characters of the regular font. This is the case for the Symbol font, Wingdings, and Webdings.

The second constructor for `FontFamily` requires a member of the `GenericFontFamilies` enumeration defined in `System.Drawing.Text`: 
Earlier I warned about possible problems creating a font based on a font family name that might not be present on some oddball system:

```java
font = new Font("Times New Roman", 24);
```

You'll sleep much better at night if you use this constructor instead:

```java
font = new Font(new FontFamily(GenericFontFamilies.Serif), 24);
```

I know, it's quite a mouthful and not nearly as elegant. But there's a shorter, equivalent version that's somewhat less verbose. The shorter version makes use of one of `FontFamily`'s static properties:

You can create a font like so:

```java
font = new Font(FontFamily.GenericSerif, 24);
```

I'll discuss the third constructor for `FontFamily` later in this chapter.

**Understanding the Design Metrics**

Programmers with experience in working with fonts in Windows or other graphical environments will probably agree that Windows Forms is really skimpy on the font metrics. So far, you've seen only three measurements that tell you anything about the height of the font: the point size (which I've emphasized is a typographical design concept only approximately related to any metrical size of the font characters); the maximum vertical extent of the font characters that you get from `MeasureString`; and the recommended line spacing, which is the value you get from `GetHeight` and, in an integer form suitable only for the video display, `Height`.

If I could have just one more font metric, it would provide me with the location of the baseline. The baseline of a font is the line above which ascenders ascend and below which descenders descend. Knowing the location of the baseline relative to the top or bottom of the characters (which is what you specify in the `DrawString` method) is necessary if you want to mix different fonts on the same line.

This information is actually available in the `FontFamily` class. It's not very obvious, and it won't work for some of the Far Eastern and Middle Eastern fonts, but it's the best that's available.

The `FontFamily` class contains four methods that let you obtain additional metrical information about the font. Each of these methods requires a `FontStyle` enumeration value as an argument:

```java
int GetEmHeight(FontStyle fs)
int GetCellAscent(FontStyle fs)
int GetCellDescent(FontStyle fs)
int GetLineSpacing(FontStyle fs)
```
These are called "design metrics" because they were originally set by the person who designed the font (or at least the TrueType version of the font). These design metrics are independent of the eventual size of the font created from this font family.

Let's look at an example. If you create a FontFamily based on Times New Roman and you call these four methods using FontStyle.Regular (or any other FontStyle value), you'll get the following numbers.

**Times New Roman Design Metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Em Height</td>
<td>2048</td>
</tr>
<tr>
<td>Ascent</td>
<td>1825</td>
</tr>
<tr>
<td>Descent</td>
<td>443</td>
</tr>
<tr>
<td>Ascent + Descent</td>
<td>2268</td>
</tr>
<tr>
<td>Line Spacing</td>
<td>2355</td>
</tr>
</tbody>
</table>

The Em Height represents the grid that the designer of the font used for specifying coordinates of the various lines and curves that define each character in the font. The value of 2048 is very common. Less common are values of 1000 or 256.

The Ascent value is the height of the font characters above the baseline (including diacritical marks), and Descent is the height below the baseline. For Times New Roman, the sum of the Ascent and Descent (which I've added to the table) represents the actual height of the font characters.

The Line Spacing value breaks down into three components: the Ascent above the baseline, the Descent below the baseline, and some extra space below the Descent. It looks like this:

```
|\̂a|q|
```

For some fonts, the Line Spacing value is greater than the sum of the cell Ascent and Descent. For other fonts, the Extra space is 0 and the sum of the Ascent and Descent equals the Line Spacing.

Do not attempt to fit the Em Height into this diagram in some way! The Em Height is simply a reference point for the other design metrics.

Let's continue the example by creating a 72-point Times New Roman font. Let us also set the PageUnit property to GraphicsUnit.Point. That means that GetHeight returns units of points regardless of the resolution of the graphics device. You'll get the following values (rounded to two decimal places):

**Times New Roman Font Metrics**

<table>
<thead>
<tr>
<th>Property or Method</th>
<th>Value (in points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>font.SizeInPoints</td>
<td>72.00</td>
</tr>
<tr>
<td>font.GetHeight(grfx)</td>
<td>82.79</td>
</tr>
</tbody>
</table>

Can this table be reconciled with the previous table showing the design metrics? Yes, because the value obtained from GetHeight is derived from those design metrics. The design metric called Em Height is equivalent to the point size of the font. If you take the ratio of 72 to 2048, you'll get 0.03515625. That's a scaling factor to convert the coordinates of the font characters to a point size.
Multiply that scaling factor by the Line Spacing metric (2355) and you get 82.79, the value returned from GetHeight.

The implication here is that we can apply that same factor separately to the Ascent and Descent design metrics to provide us with numbers we didn't have before:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Design Metric Value</th>
<th>Value for 72-Point Font (in points)</th>
<th>Property or Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Em Height</td>
<td>2048</td>
<td>72.00</td>
<td>font.SizeInPoints</td>
</tr>
<tr>
<td>Ascent</td>
<td>1825</td>
<td>64.16</td>
<td></td>
</tr>
<tr>
<td>Descent</td>
<td>443</td>
<td>15.57</td>
<td></td>
</tr>
<tr>
<td>Ascent + Descent</td>
<td>2268</td>
<td>79.73</td>
<td></td>
</tr>
<tr>
<td>Line Spacing</td>
<td>2355</td>
<td>82.79</td>
<td>font.GetHeight(grfx)</td>
</tr>
</tbody>
</table>

You can use information derived from the design metrics to position text on a baseline. Let's suppose we want the baseline to be the horizontal line in the center of the client area and we want a 144-point Times New Roman font to be positioned on that line. Here's the code to do it.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class TextOnBaseline: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TextOnBaseline());
    }
    public TextOnBaseline()
    {
        Text = "Text on Baseline";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        float yBaseline = cy / 2;
        Pen    pen    = new Pen(clr);

        // Draw the baseline across the center of the client area.

        grfx.DrawLine(pen, 0, yBaseline, cx, yBaseline);
    }
}
```
// Create a 144-point font.

Font font = new Font("Times New Roman", 144);

// Get and calculate some metrics.

float cyLineSpace = font.GetHeight(grfx);
int iCellSpace = font.FontFamily.GetLineSpacing(font.Style);
int iCellAscent = font.FontFamily.GetCellAscent(font.Style);
float cyAscent = cyLineSpace * iCellAscent / iCellSpace;

// Display the text on the baseline.

grfx.DrawString("Baseline", font, new SolidBrush(clr),
0, yBaseline - cyAscent);

The cyAscent value is the ascent for the 144-point Times New Roman font. Subtract that from the vertical coordinate of that baseline, and that's where to position text that sits on the baseline:

If you look carefully, you'll see that the rounded part of some letters actually dips below the baseline a bit, but that's normal.

Arrays of Font Families

FontFamily has one more static property and just one static method, and they are very similar and very important. Both of them return an array of FontFamily objects corresponding to the installed TrueType and OpenType fonts on the system. Here's the static property:

<table>
<thead>
<tr>
<th>FontFamily Static Property (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>FontFamily[]</td>
</tr>
</tbody>
</table>

If you call

FontFamily[] aff = FontFamily.Families;

each element of the aff array will be a FontFamily object. If you've ever done font enumeration under Windows using the Win32 API, you're probably wondering why life can't always be this easy.

The first thing you'll probably want to do with this wonderful property is list all your fonts by creating a sample font from each family. Here's a simple program that does just that.

NaiveFamiliesList.cs

//------------------------------------------------
// NaiveFamiliesList.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class NaiveFamiliesList: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new NaiveFamiliesList());
    }
    public NaiveFamiliesList()
    {
        Text = "Naive Font Families List";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        float y = 0;
        FontFamily[] aff = FontFamily.Families;

        foreach (FontFamily ff in aff)
        {
            Font font = new Font(ff, 12);
            grfx.DrawString(ff.Name, font, brush, 0, y);
            y += font.GetHeight(grfx);
        }
    }
}

The **foreach** statement goes through the elements of the **FontFamily** array. For each element, a **Font** constructor creates a 12-point font, the **DrawString** call uses that font to display the name of the font family, and the **GetHeight** call prepares the vertical coordinate for the next font.

This program might not work on your system. On mine, the **Font** constructor throws an exception for the very first font family in the array.

The problematic font file is Ahronbd.ttf, which is installed from the Windows 2000 CD-ROM as part of the support for Hebrew. The font implemented in this file is Aharoni Bold, and it's the only font in the Aharoni family (as least as far as Windows 2000 goes). What that means is that this statement will work:

```csharp
font = new Font("Aharoni", 12, FontStyle.Bold)
```

But this one won't:

```csharp
font = new Font("Aharoni", 12)
```

It won't work because it's equivalent to

```csharp
font = new Font("Aharoni", 12, FontStyle.Regular)
```
and there's no regular Aharoni font. Bummer, right?

You can deal with this problem in a couple ways. The first approach is to use a try and catch construction. Put the code in the previous foreach loop in a try block. In the catch block, display the problematic font family name with an asterisk using the Font property of the form:

```csharp
foreach (FontFamily ff in aff)
{
    try
    {
        Font font = new Font(ff, 12);
        grfx.DrawString(ff.Name, font, brush, 0, y);
        y += font.GetHeight(grfx);
    }
    catch
    {
        grfx.DrawString("* " + ff.Name, Font, brush, 0, y);
        y += Font.GetHeight(grfx);
    }
}
```

However, the general rule is that you should avoid try and catch blocks if alternative approaches are possible. The alternative approach here is the IsStyleAvailable method of the FontFamily class. Here's a better approach to listing the font families that works whether or not the Aharoni Bold font is installed.

```csharp
class BetterFamiliesList: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new BetterFamiliesList());
    }
    public BetterFamiliesList()
    {
        Text = "Better Font Families List";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        float y = 0;
        FontFamily[] aff = FontFamily.Families;
```
foreach (FontFamily ff in aff)
{
    if (ff.IsStyleAvailable(FontStyle.Regular))
    {
        Font font = new Font(ff, 12);
        grfx.DrawString(ff.Name, font, brush, 0, y);
        y += font.GetHeight(grfx);
    }
    else
    {
        grfx.DrawString("* " + ff.Name, Font, brush, 0, y);
        y += Font.GetHeight(grfx);
    }
}
}

If you have a large number of fonts installed on your system, however, this approach won’t let you see them all, even if you have a big monitor and even if you print the list.

Short of adding scroll bars, a much better approach is to format the list into columns. That's what the following program does. And if this approach still isn’t sufficient for all your fonts, try changing the iPointSize field to 10, 8, or 6.

FamiliesList.cs
//-------------------------------------------
// FamiliesList.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FamiliesList: PrintableForm
{
    const int iPointSize = 12;

    public new static void Main()
    {
        Application.Run(new FamiliesList());
    }
    public FamiliesList()
    {
        Text = "Font Families List";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
float x = 0, y = 0, fMaxWidth = 0;
FontFamily[] aff = GetFontFamilyArray(grfx);

foreach (FontFamily ff in aff)
{
    Font font = CreateSampleFont(ff, iPointSize);
    SizeF sizef = grfx.MeasureString(ff.Name, font);
    fMaxWidth = Math.Max(fMaxWidth, sizef.Width);
}

foreach (FontFamily ff in aff)
{
    Font font = CreateSampleFont(ff, iPointSize);
    float fHeight = font.GetHeight(grfx);

    if (y > 0 && y + fHeight > cy)
    {
        x += fMaxWidth;
        y = 0;
    }
    grfx.DrawString(ff.Name, font, brush, x, y);

    y += fHeight;
}

protected virtual FontFamily[] GetFontFamilyArray(Graphics grfx)
{
    return FontFamily.Families;
}

Font CreateSampleFont(FontFamily ff, float fPointSize)
{
    if (ff.IsStyleAvailable(FontStyle.Regular))
        return new Font(ff, fPointSize);

    else if (ff.IsStyleAvailable(FontStyle.Bold))
        return new Font(ff, fPointSize, FontStyle.Bold);

    else if (ff.IsStyleAvailable(FontStyle.Italic))
        return new Font(ff, fPointSize, FontStyle.Italic);

    else
        return Font;
}
The `DoPage` method has two `foreach` loops. The first determines the width of each font family name displayed in a sample font from that family and saves the maximum width; the second `foreach` loop uses that maximum width to display multiple columns.

Notice the `CreateSampleFont` method down at the bottom of the class. I use this in the `DoPage` method instead of using the `Font` constructor. `CreateSampleFont` uses the `IsStyleAvailable` method to determine whether to create a regular, bold, or italic font. This approach succeeds in displaying a sample Aharoni font where the previous programs did not. Here's a nonmaximized version of the program running on my machine:

![Sample Fonts](image)

If you've installed some Far Eastern or Middle Eastern fonts, you'll notice that the line spacing seems to be excessively large for the displayed text. That's because these fonts are designed for displaying alphabets other than Latin.

This FamiliesList program isolates the statement that obtains the array of `FontFamily` objects in a short method I named `GetFontFamilyArray`. I did that so that in the next program I can easily demonstrate the use of the only static method implemented in `FontFamily`. This static method is similar to the `Families` property except that it has an argument of type `Graphics`:

**FontFamily Static Method**

```csharp
FontFamily[] GetFamilies(Graphics grfx)
```

The idea here is that different graphics output devices might have different fonts installed; in particular, some printers have built-in fonts that can't be displayed on the screen. This program overrides the `GetFontFamilyArray` to get the font families from `GetFamilies` rather than `Families`.

**GetFamiliesList.cs**

```csharp
// GetFamiliesList.cs © 2001 by Charles Petzold
//----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class GetFamiliesList: FamiliesList
{
    public new static void Main()
    {
        Application.Run(new GetFamiliesList());
    }
    public GetFamiliesList()
    {
        Text = "Font GetFamilies List";
    }
```
protected override FontFamily[] GetFontFamilyArray(Graphics grfx)
{
    return FontFamily.GetFamilies(grfx);
}

At least for my printer, GetFamilies returns the same array as Families. Perhaps in a later version of Windows Forms we'll see more support for printer-specific fonts.

Font Collections

The Families property and GetFamilies method of the FontFamily class are not the only way to get an array of font families. The System.Drawing.Text namespace has an abstract FontCollection class from which two other classes are derived: InstalledFontCollection and PrivateFontCollection.

FontCollection implements just one property:

<table>
<thead>
<tr>
<th>FontCollection Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>FontFamily[]</td>
</tr>
</tbody>
</table>

This property—which is not defined as static—is inherited by both InstalledFontCollection and PrivateFontCollection. The following program overrides the GetFontFamilyArray method in the FamiliesList program with code that creates an instance of the InstalledFontCollection class and uses the Families property to get the array of font families.

InstalledFontsList.cs
//-------------------------------------------------
// InstalledFontsList.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;

class InstalledFontsList: FamiliesList
{
    public new static void Main()
    {
        Application.Run(new InstalledFontsList());
    }
    public InstalledFontsList()
    {
        Text = "InstalledFontCollection List";
    }
    protected override FontFamily[] GetFontFamilyArray(Graphics grfx)
    {
        FontCollection fc = new InstalledFontCollection();
        return fc.Families;
    }
}
}
This program produces the same output as the FamiliesList program.

When you create an instance of the `PrivateFontCollection` class, it initially contains no font families. You add fonts to the collection by using the following two methods:

**PrivateFontCollection Methods (selection)**

```csharp
void AddFontFile(string strFilename)
void AddMemoryFont(IntPtr pFont, int iLength)
```

This facility is used by applications that include their own specialized font files. After creating a `PrivateFontCollection` object and calling these two methods, the application can then use the `Families` property to obtain an array of `FontFamily` objects suitable for creating `Font` objects. Or (since your program probably knows what font families are included in this collection) you can use the third `FontFamily` constructor in the list on page 387 to create `FontFamily` objects based on font files in this collection.

**Variations on DrawString**

We've already encountered the six variations of the `DrawString` method:

**Graphics DrawString Methods**

```csharp
void DrawString(string str, Font font, Brush brush, PointF ptf)
void DrawString(string str, Font font, Brush brush, float x, float y)
void DrawString(string str, Font font, Brush brush, RectangleF rectf)
void DrawString(string str, Font font, Brush brush, PointF ptf,
               StringFormat sf)
void DrawString(string str, Font font, Brush brush, float x, float y,
               StringFormat sf)
void DrawString(string str, Font font, Brush brush, RectangleF rectf,
               StringFormat sf)
```

The versions using a `PointF` structure are identical to the versions using the two `float` values. It's just two different ways of specifying the same starting point for the string. All four of the `DrawString` overloads that use a `PointF` or two `float` values generally display a single line of text. However, if the text contains line feed characters ('\n' or '\x000A'), the text that follows is displayed one line lower.

The two versions of `DrawString` that use a `RectangleF` argument wrap text that is too wide to fit within the rectangle width. If a single word is too wide for the rectangle, the method will fit as much of the word as possible and then display the remainder of the word on the next line.

These versions of `DrawString` properly recognize Unicode character '\x00A0', the No-Break Space. You use a no-break space instead of a regular space when text wrapped at the space would look peculiar, for example, in the following string:

"World War\x00A0I"

In this case, if the `I` doesn't fit at the end of a line, the `DrawString` method would break the line after the word `World`.

These versions of `DrawString` do **not** properly recognize Unicode character '\x00AD', the Soft Hyphen. Customarily, you insert soft hyphens at the syllable breaks in long words:

"ex\x00ADtraor\x00ADdi\x00ADnary"
In theory, if the text formatter can break the line at the hyphen, it will do so and display the hyphen. If not, the hyphen won't be displayed. The `DrawString` method displays these hyphens regardless, and even breaks a line before the hyphen.

These versions of `DrawString` also do not properly handle Unicode character '\x2011', the Non-Breaking Hyphen, which you use in words like this:

"T\x2011shirt"

The T followed by a hyphen would look odd at the end of a line. Some TrueType fonts don't include this character. In those fonts that do include the character (such as Lucida Sans Unicode), the `DrawString` method seems to avoid breaking a line on the Non-Breaking Hyphen, but it inserts some extra space following the hyphen.

Because the word-wrapping ability of the `DrawString` call seems so powerful, it's important to understand its limitations. For example, you may find `DrawString` perfect for displaying a particular block of text in your application, but with just one little problem: there's a word in the text that needs to be bold or italic. Well, you can't do it. The `Font` argument to `DrawString` determines the font for the entire block of text. (Interestingly enough, however, there is a way you can underline selected text in a `DrawString` call. I'll demonstrate how to do that in the UnderlinedText program later in this chapter.)

Another limitation: as you know, you can use the `Alignment` property of the `StringFormat` class to control the horizontal alignment of text you display in a rectangle. You can align a paragraph on the left edge of the rectangle, you can align a paragraph on the right, and you can center the lines of the paragraph within the rectangle. But you can't justify the rectangle. You can't instruct `DrawString` to insert extra space between the words so that the left margin and the right margin are both even.

If you need to do either of these jobs, you have two choices: you can either write your own text-formatting logic or make use of the `RichTextBox` control that has many more built-in formatting options than `DrawString` does. I discuss `RichTextBox` in Chapter 18.

**Anti-Aliased Text**

In Chapter 5, I showed how the `SmoothingMode` and `PixelOffsetMode` properties of the `Graphics` class govern anti-aliasing of lines and curves. Windows can also use anti-aliasing for the display of text. This feature is under user control. You can turn it on and off by checking the option Smooth Edges Of Screen Fonts in the Effects tab of the Display Properties dialog box.

If you want, you can override the user's preferences by setting the `TextRenderingHint` property of the `Graphics` class:

<table>
<thead>
<tr>
<th>Graphics Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>TextRenderingHint</td>
</tr>
</tbody>
</table>

`TextRenderingHint` is also an enumeration defined in the `System.Drawing.Text` namespace:

<table>
<thead>
<tr>
<th>TextRenderingHint Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>SystemDefault</td>
</tr>
<tr>
<td>SingleBitPerPixelGridFit</td>
</tr>
<tr>
<td>SingleBitPerPixel</td>
</tr>
<tr>
<td>AntiAlias</td>
</tr>
<tr>
<td>AntiAliasGridFit</td>
</tr>
<tr>
<td>ClearTypeGridFit</td>
</tr>
</tbody>
</table>

ClearType is a technology similar to anti-aliasing but which takes advantage of the arrangement of color dots on LCD displays.[1]
The following program demonstrates the use of all six of these enumeration values.

**AntiAliasedText.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;

class AntiAliasedText: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new AntiAliasedText());
    }
    public AntiAliasedText()
    {
        Text = "Anti-Aliased Text";
        Font = new Font("Times New Roman", 12);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        string str = "A ";
        int cxText = (int) grfx.MeasureString(str, Font).Width;

        for (int i = 0; i < 6; i++)
        {
            grfx.TextRenderingHint = (TextRenderingHint)i;
            grfx.DrawString(str, Font, brush, i * cxText, 0);
        }
    }
}
```

The program displays a capital A six times using the six `TextRenderingHint` values. You can copy an image of this program into the clipboard by using Alt+Print Scrn and then paste the image into a graphics paint or photo program, where you can blow it up like so:

```
A A A A A A
```

Depending on your settings, the first (`SystemDefault`) will match either the second (`SingleBitPerPixelGridFit`) or the fourth (`AntiAlias`), or if you're using ClearType on an LCD display the first could match the sixth (`ClearTypeGridFit`). Neither the second nor the third enumeration value (`SingleBitPerPixel`) causes any anti-aliasing of text to be performed. However, notice one or more blank pixels in the third capital A. The character rasterizer is using a stricter criteria for determining whether or not a pixel should be colored.
The fourth and fifth *(AntiAliasGridFit)* examples use anti-aliasing. Pixels are colored with a darker gray shade depending on the extent to which they intersect the theoretical outlined character.

For *ClearTypeGridFit*, pixels to the left and right of the strokes are in different colors based on the horizontal organization of color dots on LCD displays. You should not use ClearType on regular CRTs. Some LCD displays allow users to rotate them 90 degrees and use them in portrait mode. ClearType won’t work in that case either.

It turns out that this technique has been discovered and rediscovered over a period of more than two decades. See Steve Gibson’s discussion of the technology and its history on his Web site at [http://grc.com/cleartype.htm](http://grc.com/cleartype.htm).

### Measuring the String

Since Chapter 2, we’ve been using *MeasureString* to determine the size of a text string for accurate positioning. *MeasureString* also has another use. As you’ve probably noticed, the *DrawString* method displays text without also erasing the background of the destination rectangle. (This isn’t the default behavior in Windows GDI.) If you do need to erase the background, you can use the coordinate point at which you’ll be drawing the text combined with the *SizeF* returned from *MeasureString* in a *FillRectangle* call.

The *MeasureString* method comes in seven versions:

#### Graphics *MeasureString* Methods

```csharp
SizeF MeasureString(string str, Font font);
SizeF MeasureString(string str, Font font, int iWidth);
SizeF MeasureString(string str, Font font, SizeF sizef);
SizeF MeasureString(string str, Font font, int iWidth,
    StringFormat strfmt);
SizeF MeasureString(string str, Font font, SizeF sizef,
    StringFormat strfmt);
SizeF MeasureString(string str, Font font, PointF ptfOrigin,
    StringFormat strfmt);
SizeF MeasureString(string str, Font font, SizeF sizef,
    StringFormat strfmt,
    out int iCharacters, out int iLines);
```

We’ve been using the first version of *MeasureString* for quite some time. It returns the width and height of the specified string as displayed using the specified font. The *Height* property of the *SizeF* object returned from the method is often equal to the value returned from the *GetHeight* method of the *Font* class, but it could be a multiple of the *GetHeight* value if the text includes line feed characters.

The second version of *MeasureString* includes a third argument that indicates a text width. This version is useful if you’ll be displaying the string using the *RectangleF* version of *DrawString* and want wrapping to occur. The *Width* property of the *SizeF* object returned from *MeasureString* is always less than or equal to the *iWidth* argument; the *Height* property, when divided by the *GetHeight* value, equals the number of lines.

The third version of *MeasureString* has an actual *SizeF* argument, indicating both a width and a height. If the *Width* property of this *SizeF* argument is the same as the *iWidth* argument used in the second version of *MeasureString* and if the *Height* property is sufficient for all the lines of text in the string, the return value of this version will be the same as the second version. Otherwise, the *Height* property of the returned *SizeF* object will equal the *Height* property of the *SizeF* argument, and the *Width* property of the returned *SizeF* object will indicate the maximum width of the text that can fit in that size rectangle.
The fourth, fifth, and sixth versions are similar to the second and third except that they include a StringFormat argument. If you’ll be using a StringFormat argument in the DrawString call, you should also use one in the MeasureString call.

The final version of MeasureString has two arguments that return additional information to the application. These indicate the number of characters and lines of text that a DrawString call will display when passed a RectangleF structure of the same size as the SizeF structure and the same StringFormat object.

Calling MeasureString with these arguments is extremely useful when you need to use multiple DrawString calls to display a single block of text. For example, suppose you want to use DrawString to display text to the printer but the text is too long for a single page. You use MeasureString to determine how much can fit on the first page and then start the second page with a new text string based on information returned from MeasureString. I’ll demonstrate the use of this version of MeasureString in the TextColumns program toward the end of this chapter.

The StringFormat Options

The DrawString and MeasureString methods can optionally include an argument that is an object of type StringFormat. This argument offers you many different—sometimes subtle and sometimes not so subtle—variations in the display of text. You can create a StringFormat object by using one of the following constructors:

```
StringFormat()  // Creates an empty StringFormat object.
StringFormat(StringFormat strfmt)  // Clones an existing StringFormat object.
StringFormat(StringFormatFlags sff)  // Creates a StringFormat object based on the specified flags.
StringFormat(StringFormatFlags sff, int iLanguage)  // Creates a StringFormat object with specified flags and language.
```

The second version essentially clones an existing StringFormat object; the third and fourth versions create a StringFormat object based on a combination of StringFormatFlags enumeration values. The StringFormatFlags enumeration is also used in setting the FormatFlags property of StringFormat:

```
StringFormatProperties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringFormatFlags</td>
<td>FormatFlags</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

The StringFormatFlags enumeration is a series of bit flags:

```
StringFormatFlags Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectionRightToLeft</td>
<td>1</td>
</tr>
<tr>
<td>DirectionVertical</td>
<td>2</td>
</tr>
<tr>
<td>FitBlackBox</td>
<td>4</td>
</tr>
<tr>
<td>DisplayFormatControl</td>
<td>32</td>
</tr>
<tr>
<td>NoFontFallback</td>
<td>1024</td>
</tr>
<tr>
<td>MeasureTrailingSpaces</td>
<td>2048</td>
</tr>
<tr>
<td>NoWrap</td>
<td>4096</td>
</tr>
<tr>
<td>LineLimit</td>
<td>8192</td>
</tr>
<tr>
<td>NoClip</td>
<td>16384</td>
</tr>
</tbody>
</table>
```
I discuss some of these flags in this book in connection with certain programs. I use MeasureTrailingSpaces in the TypeAway program in Chapter 6, and in the BoldAndItalicTighter program coming up shortly. I demonstrate NoWrap and NoClip in the TrimmingTheText program later in this chapter.

When you create a new StringFormat object using the default constructor, the FormatFlags property is set to 0. Notice that these enumeration values are single bits, so you can combine them with the C# logical OR operator. For example,

```csharp
StringFormat strfmt = new StringFormat(StringFormatFlags.DirectionVertical | StringFormatFlags.NoClip);
```

When you're setting the FormatFlags property, I strongly recommend that you get into the habit of using the |= operator:

```csharp
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```

That way, you'll never accidentally turn off one of the other flags that you may have set earlier.

Besides using one of the constructors, you can obtain a StringFormat object by using one of the following static properties:

<table>
<thead>
<tr>
<th>StringFormat Static Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>StringFormat</td>
</tr>
<tr>
<td>StringFormat</td>
</tr>
</tbody>
</table>

If you examine the properties of the StringFormat objects returned by these static properties, you'll discover that GenericDefault returns a StringFormat object that is the same as that created by the default constructor. The GenericTypographic property returns an object that has the FitBlackBox, LineLimit, and NoClip flags set as well as a different value for the Trimming property, which I'll discuss later in this chapter.

However, the StringFormat object that GenericTypographic returns has an additional effect on DrawString and MeasureString that is not revealed by the public StringFormat properties and flags. This additional effect is what I'll tackle next.

**Grid Fitting and Text Fitting**

The text-handling portion of GDI+ is designed to be device independent. In practical terms, MeasureString returns a text dimension that is independent of the output device. If you set the same page transform on both the screen and the printer, MeasureString returns identical values for any particular text string and font. This consistency makes it comparatively easy to format text on the screen that will look the same when printed on the printer.

However desirable the goal of WYSIWYG may be, it's not easy to realize in real life. The problem is pixels. When the characters of an outline font are rasterized, the original floating-point coordinates must be rounded to discrete pixels. Such grid fitting, as it's called, requires hints that prevent this rounding from destroying the legibility of the font. The two vertical strokes on a capital H must be the same width, for example. Even for small point sizes, these two strokes must be at least 1 pixel wide and must be separated by a pixel. (If the point size is very small compared to the resolution of the output device, such requirements can be abandoned because the text wouldn't be legible anyway.)

In some cases, particularly for small point sizes on low-resolution devices (such as the video display), grid fitting can cause rendered characters to be noticeably larger than their theoretical size. String a bunch of these characters together (for example, lower case Arial Is), and you could end up with a text string substantially larger on the screen than on the printer. (See the article at http://www.gotdotnet.com/team/windowsforms/gdipitext.aspx for more-detailed examples of this problem.)

When you use DrawString and MeasureString to concatenate pieces of text (such as in the BoldAndItalic programs shown earlier in this chapter), would you prefer that the resultant output has extra space between the pieces or that the pieces of text overlap somewhat? I think you'd agree that
overlapping text is the less desirable alternative. To prevent overlapping text, the `DrawString` and `MeasureString` methods have been deliberately finagled to include a little extra space. Thus, if the rasterizer requires more space to render a particular font, that space is available.

By default, the `SizeF` object returned from `MeasureString` has a `Height` property 1/8 em greater than what is theoretically necessary and a `Width` property 1/3 em more than the theoretical width in addition to a small percentage increase. (Remember that an em is equal to the point size of the font. For example, for a 24-point font, 1/3 em equals 8 points.) By default, the `DrawString` method begins displaying text 1/6 em beyond the specified vertical coordinate. In effect, `MeasureString` indicates a rectangle that is wider than the theoretical text string by 1/6 em on the right and left.

And that's why the BoldAndItalic program and, more demonstrably, the BoldAndItalicBigger program, have superfluous padding between the concatenated pieces of text.

Keep in mind that the grid-fitting problem affects only fonts with small point sizes displayed on low-resolution devices. To achieve device independence, however, the extra padding built into `DrawString` and `MeasureString` must be the same for both low-resolution and high-resolution devices. And the padding must be proportional for small fonts and large fonts. For a 720-point font, `MeasureString` must return a text size 100 times greater than for a 7.2-point font.

What do you do if you don't want this extra space? You simply use a `StringFormat` object based on `StringFormat.GenericTypographic`. Here's a version of the BoldAndItalic program that uses such a `StringFormat` object.

`BoldAndItalicTighter.cs`

```csharp
//---------------------------------------------------
// BoldAndItalicTighter.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;

class BoldAndItalicTighter: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new BoldAndItalicTighter());
    }

    public BoldAndItalicTighter()
    {
        Text = "Bold and Italic (Tighter)";
        Font = new Font("Times New Roman", 24);
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        const string str1 = "This is some ";
        const string str2 = "bold";
        const string str3 = " text, and this is some ";
        const string str4 = "italic";
        const string str5 = " text.";
        Brush brush = new SolidBrush(clr);
    }
}
```
Font fontRegular = Font;
Font fontBold = new Font(fontRegular, FontStyle.Bold);
Font fontItalic = new Font(fontRegular, FontStyle.Italic);
PointF ptf = new PointF(0, 0);
StringFormat strfmt = StringFormat.GenericTypographic;
strfmt.FormatFlags |= StringFormatFlags.MeasureTrailingSpaces;

grfx.DrawString(str1, fontRegular, brush, ptf, strfmt);
ptf.X += grfx.MeasureString(str1, fontRegular, ptf, strfmt).Width;

grfx.DrawString(str2, fontBold, brush, ptf, strfmt);
ptf.X += grfx.MeasureString(str2, fontBold, ptf, strfmt).Width;

grfx.DrawString(str3, fontRegular, brush, ptf, strfmt);
ptf.X += grfx.MeasureString(str3, fontRegular, ptf, strfmt).Width;

grfx.DrawString(str4, fontItalic, brush, ptf, strfmt);
ptf.X += grfx.MeasureString(str4, fontItalic, ptf, strfmt).Width;

grfx.DrawString(str5, fontRegular, brush, ptf, strfmt);
}
}

Notice that the program also sets the MeasureTrailingSpaces flag. The result looks just fine:

This is some **bold** text, and this is some *italic* text.

I can get away with using the GenericTypographic object in BoldAndItalicTighter because I know that the font is large enough that a few pixels here and there won’t make a difference. If you want to use GenericTypographic in small font sizes on the video display, you should also enable anti-aliasing. Anti-aliasing avoids grid fitting approximations because each pixel is colored based on its intersection with the theoretical outline.

**Horizontal and Vertical Alignment**

Our first encounter with the StringFormat class was back in Chapter 3, where we used it to center text in a form’s client area. The two StringFormat properties that affect the alignment of text are shown here:

<table>
<thead>
<tr>
<th>StringFormat Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>StringAlignment</td>
</tr>
<tr>
<td>StringAlignment</td>
</tr>
</tbody>
</table>
Both these properties are of type `StringAlignment`, which is an enumeration consisting of just three members:

**StringAlignment Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near</td>
<td>0</td>
<td>Usually left or top</td>
</tr>
<tr>
<td>Center</td>
<td>1</td>
<td>Always the center</td>
</tr>
<tr>
<td>Far</td>
<td>2</td>
<td>Usually right or bottom</td>
</tr>
</tbody>
</table>

The alignment values work a little differently depending on whether you specify a `PointF` object or a `RectangleF` object in the `DrawString` call. Let's take a look at the `RectangleF` version of `DrawString` first. The following program uses the client area rectangle in nine `DrawString` calls, each of which uses a different combination of `Alignment` and `LineAlignment` properties. Just to make it interesting, the text I'm displaying with each `DrawString` call has an embedded line feed character.

**StringAlignmentRectangle.cs**

```csharp
//-------------------------------------------------------
// StringAlignmentRectangle.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class StringAlignmentRectangle : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new StringAlignmentRectangle());
    }
    public StringAlignmentRectangle()
    {
        Text = "String Alignment (RectangleF in DrawString)";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        RectangleF rectf = new RectangleF(0, 0, cx, cy);
        string[] strAlign = { "Near", "Center", "Far" };
        StringFormat strfmt = new StringFormat();

        for (int iVert = 0; iVert < 3; iVert++)
        for (int iHorz = 0; iHorz < 3; iHorz++)
        {
            strfmt.LineAlignment = (StringAlignment)iVert;
            strfmt.Alignment = (StringAlignment)iHorz;

            grfx.DrawString(String.Format("LineAlignment = {0}\nAlignment = {1}"),
```
```
The three possible values of the **Alignment** property cause the text to be left aligned, centered, or right aligned in the rectangle. The three possible values of **LineAlignment** cause the text to be displayed at the top, center, or bottom of the rectangle:

![Alignment Diagram](image)

Such a nice, neat, well-ordered display isn't possible when you use the same PointF object in multiple calls to DrawString. Some of the combinations would overlap with others. Let's instead look at a few at a time.

This program sets the PointF object in the DrawString call to the center of the client area but uses only four combinations of the Alignment and LineAlignment properties.

**StringAlignmentPoint.cs**

```csharp
//---------------------------------------------------
// StringAlignmentPoint.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class StringAlignmentPoint: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new StringAlignmentPoint());
    }
    public StringAlignmentPoint()
    {
        Text = "String Alignment (PointF in DrawString)";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush    = new SolidBrush(clr);
        Pen pen        = new Pen(clr);
        string[] strAlign = { "Near", "Center", "Far" };  // strAlign[iVert], strAlign[iHorz]),
        Font, brush, rectf, strfmt);
    }
}
```
StringFormat strfmt  = new StringFormat();

grfx.DrawLine(pen, 0, cy / 2, cx, cy / 2);
grfx.DrawLine(pen, cx / 2, 0, cx / 2, cy);

for (int iVert = 0; iVert < 3; iVert += 2)
for (int iHorz = 0; iHorz < 3; iHorz += 2)
{
    strfmt.LineAlignment = (StringAlignment)iVert;
    strfmt.Alignment     = (StringAlignment)iHorz;

    grfx.DrawString(
        String.Format("LineAlignment = {0}\nAlignment = {1}",
                      strAlign[iVert], strAlign[iHorz]),
        Font, brush, cx / 2, cy / 2, strfmt);
}

Notice the two for statements: the iVert and iHorz variables end up being set to only 0 and 2, and the program uses only four combinations of the Alignment and LineAlignment properties to create a display that looks like this:

The PointF object passed to DrawString is the center of the client area. The DrawString call positions the two lines of text relative to this coordinate depending on the settings of the Alignment and LineAlignment properties.

If you change the first for statement in this program to
for (int iVert = 1; iVert < 3; iVert += 2)

and recompile, you can see how a LineAlignment property set to StringAlignmentCenter causes the vertical coordinate passed to DrawString to specify the center of the text. In this case, that's the vertical center of the two text lines:
If instead you change the second for statement to
for (int iHorz = 1; iHorz < 3; iHorz += 2)
the two lines of text are centered around the horizontal coordinate:

If you change both for loops as just shown, you'll get the final case. The two lines of text are centered horizontally and vertically around the midpoint of the client area.
The Hotkey Display

The HotkeyPrefix property of StringFormat determines how the DrawString call interprets ampersands:

<table>
<thead>
<tr>
<th>StringFormat Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>HotkeyPrefix</td>
</tr>
</tbody>
</table>

How the DrawString call interprets ampersands? That might sound a little odd until you realize that embedded ampersands have a special meaning in the text used in menu items, buttons, and other controls. The ampersand indicates that the character that follows is to be underlined and that the character is to function as a keyboard shortcut.

You set the HotkeyPrefix property to one of the following HotkeyPrefix enumeration values defined in the System.Drawing.Text namespace:

<table>
<thead>
<tr>
<th>HotkeyPrefix Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Show</td>
</tr>
<tr>
<td>Hide</td>
</tr>
</tbody>
</table>

By default, ampersands aren't treated special and are simply displayed as ampersands. The Show value suppresses the ampersand and forces the next character to be underlined. The Hide value suppresses the ampersand but doesn't underline the next character.

Even if you're not displaying text in menus or controls, you can use this property to underline specific letters or words that appear in blocks of text you pass to the DrawString call. Here's a program that demonstrates this technique.

UnderlinedText.cs

```csharp
using System;
using System.Drawing;
```
using System.Drawing.Text;
using System.Windows.Forms;

class UnderlinedText: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new UnderlinedText());
    }
    public UnderlinedText()
    {
        Text = "Underlined Text Using HotkeyPrefix";
        Font = new Font("Times New Roman", 14);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        string str = "This is some underlined text!";
        StringFormat strfmt = new StringFormat();
        strfmt.HotkeyPrefix = HotkeyPrefix.Show;

        grfx.DrawString(str, Font, new SolidBrush(clr), 0, 0, strfmt);
    }
}

The string in this program listing doesn't appear very attractive, but the results look quite nice:

![Underlined Text Using HotkeyPrefix](image)

It's too bad there's not also some kind of facility to italicize or boldface words in a text block passed to DrawString.

I use HotkeyPrefix for what it's designed for in the OwnerDrawMenu program in Chapter 14.

**A Clip and a Trim**

When you use the RectangleF version of DrawString, you're defining not only a right margin that governs text wrapping but also a bottom margin that limits the total amount of text that can be displayed.

What happens if the text is too long to fit in the rectangle?

Let's look at the default case first—when you don't include a StringFormat object as the last argument to DrawString. If the height of the rectangle is an integral multiple of the line-spacing height, an integral number of lines of text can fit in the rectangle. The last line of displayed text will contain as many characters as can fit in the rectangle width. Notice I said characters that can fit—not necessarily complete words. To let you explore the way this works, here's a version of the
HuckleberryFinn program from Chapter 3 that restricts the text to half the client area width and height.

HuckleberryFinnHalfHeight.cs

//--------------------------------------------------------
// HuckleberryFinnHalfHeight.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HuckleberryFinnHalfHeight: Form
{
    public static void Main()
    {
        Application.Run(new HuckleberryFinnHalfHeight());
    }
    public HuckleberryFinnHalfHeight()
    {
        Text = """The Adventures of Huckleberry Finn""";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        int cx = ClientSize.Width;
        int cy = ClientSize.Height;
        Pen pen = new Pen(ForeColor);
        Rectangle rect = new Rectangle(0, 0, cx / 2, cy / 2);

        grfx.DrawString(
            "You don't know about me, without you " +
            "have read a book by the name of "", ""The " +
            "Adventures of Tom Sawyer," but that " +
            "ain't no matter. That book was made by " +
            "Mr. Mark Twain, and he told the truth, " +
            "mainly. There was things which he " +
            "stretched, but mainly he told the truth. " +
            "That is nothing. I never seen anybody " +
            "but lied, one time or another, without " +
            "it was Aunt Polly, or the widow, or " +
            "maybe Mary. Aunt Polly\x2014Tom's Aunt " +
            "Polly, she is\x2014and Mary, and the Widow " +
            "Douglas, is all told about in that book" +
            "\x2014which is mostly a true book; with " +
        ")
"some stretchers, as I said before.",
Font, new SolidBrush(ForeColor), rect);

grfx.DrawLine(pen, 0, cy / 2, cx / 2, cy / 2);
grfx.DrawLine(pen, cx / 2, 0, cx / 2, cy / 2);

The program also draws lines to indicate the rectangle in which the text is being displayed.

When the display rectangle isn't sufficient for the entire paragraph, you'll notice that the last line displayed in the client area may end with an incomplete word:

As you make the display rectangle taller, there comes a point when DrawString decides that there's enough room to display an additional line of text. It's sooner than you might think! DrawString displays an additional line of text when the height of the rectangle exceeds an additional 25 percent of the line-spacing height. The last line is clipped to the interior of the rectangle.
Although the last line is mostly clipped, you can see that this new last line concludes with another partial word—the first two letters of the word *mainly*.

You can alter this default behavior by using the *Trimming* property of the *StringFormat* class:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringTrimming</td>
<td>Trimming</td>
<td>set/get</td>
</tr>
</tbody>
</table>

The *Trimming* property determines how the last line of the text is terminated when you use the *RectangleF* version of *DrawString* and the rectangle isn't large enough to fit all the text. The value is a member of the *StringTrimming* enumeration used solely in connection with this property:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>As if no bottom margin</td>
</tr>
<tr>
<td>Character</td>
<td>1</td>
<td>End on character</td>
</tr>
<tr>
<td>Word</td>
<td>2</td>
<td>End on word</td>
</tr>
<tr>
<td>EllipsisCharacter</td>
<td>3</td>
<td>End on character with ellipsis (...)</td>
</tr>
<tr>
<td>EllipsisWord</td>
<td>4</td>
<td>End on word with ellipsis (...)</td>
</tr>
<tr>
<td>EllipsisPath</td>
<td>5</td>
<td>Ellipsis preceding last directory</td>
</tr>
</tbody>
</table>

Here's a program that illustrates the effect of these values.

**TrimmingTheText.cs**

```csharp
//----------------------------------------------
// TrimmingTheText.cs © 2001 by Charles Petzold
//----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class TrimmingTheText: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TrimmingTheText());
    }
    public TrimmingTheText()
    {
        Text = "Trimming the Text";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush       brush  = new SolidBrush(clr);
        float       cyText = Font.GetHeight(grfx);
        float       cyRect = cyText;
        RectangleF  rectf  = new RectangleF(0, 0, cx, cyRect);
```
string str = "Those who profess to favor freedom and " + "yet depreciate agitation. . .want " + "crops without plowing up the ground, " + "they want rain without thunder and " + "lightning. They want the ocean without " + "the awful roar of its many waters. " + "\x2014 Frederick Douglass";

StringFormat strfmt = new StringFormat();

strfmt.Trimming = StringTrimming.Character;
grfx.DrawString("Character: " + str, Font, brush, rectf, strfmt);

rectf.Offset(0, cyRect + cyText);

strfmt.Trimming = StringTrimming.Word;
grfx.DrawString("Word: " + str, Font, brush, rectf, strfmt);

rectf.Offset(0, cyRect + cyText);

strfmt.Trimming = StringTrimming.EllipsisCharacter;
grfx.DrawString("EllipsisCharacter: " + str, Font, brush, rectf, strfmt);

rectf.Offset(0, cyRect + cyText);

strfmt.Trimming = StringTrimming.EllipsisWord;
grfx.DrawString("EllipsisWord: " + str, Font, brush, rectf, strfmt);

rectf.Offset(0, cyRect + cyText);

strfmt.Trimming = StringTrimming.EllipsisPath;
grfx.DrawString("EllipsisPath: " + Environment.GetFolderPath(Environment.SpecialFolder.Personal), Font, brush, rectf, strfmt);

rectf.Offset(0, cyRect + cyText);

strfmt.Trimming = StringTrimming.None;
grfx.DrawString("None: " + str, Font, brush, rectf, strfmt);
This program defines a RectangleF object sufficient in height for a single line of text. Using the six possible StringTrimming values, the program displays some text. This text is a quotation from Frederick Douglass for all values except StringTrimming.EllipsisPath, in which case the program uses the static Environment.GetFolderPath method to obtain the path of your My Documents folder. You can adjust the width of the window and examine how it affects the text. Here’s a typical display:

<table>
<thead>
<tr>
<th>Trimming the Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character: Those who profess to favor freedom,</td>
</tr>
<tr>
<td>Word: Those who profess to favor freedom,</td>
</tr>
<tr>
<td>EllipsisCharacter: Those who profess to favor freedom,</td>
</tr>
<tr>
<td>EllipsisWord: Those who profess to favor freedom,</td>
</tr>
<tr>
<td>EllipsisPath: C:\Documents \My Documents</td>
</tr>
<tr>
<td>None: Those who profess to favor freedom,</td>
</tr>
</tbody>
</table>

Both the EllipsisCharacter and the EllipsisWord members of StringTrimming cause an ellipsis (…) to be displayed at the end of the string, indicating that not enough room was available to display it. Both Character and EllipsisCharacter can result in a partial word being displayed.

The EllipsisPath member of StringTrimming is specifically for displaying file path names. Notice that the ellipsis is embedded in the middle of the text to favor the display of the beginning and the end of the path specification.

Just offhand, the None member appears to be the same as Word, but we’re not quite finished with this exercise, and you’ll find out why I put this one down at the bottom.

You can go into the program and change the calculation of cyRect from

```csharp
float cyRect = cyText;
```

to

```csharp
float cyRect = 1.5f * cyText;
```

Now recompile and run the new version. Here’s a typical display:
The `DrawString` method is now displaying two lines of text, and although you can’t see much of the second line, that’s the line being affected by the `Trimming` property.

At this point, you might want to see what effect the `NoWrap` flag of the `StringFormatFlags` enumeration has on this display. Add the following statement after the creation of the `StringFormat` object but before any `DrawString` calls:

```csharp
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```

Or you can put the flag in the `StringFormat` constructor:

```csharp
StringFormat strfmt = new StringFormat(StringFormatFlags.NoWrap);
```

As the name implies, the `NoWrap` flag suppresses the line-wrapping facility of `DrawString`:

The text still ends at the right margin of the rectangle, however. What you might not see clearly (unless you change the right margin of the rectangle to something less than the width of the client area) is that the `StringTrimming.None` case causes the last letter to be partially truncated at the right margin. This is the only case we’ve seen so far in which a partial letter is displayed.

Now get rid of that `NoWrap` flag. Or rather, replace it with the `NoClip` flag:

```csharp
strfmt.FormatFlags = StringFormatFlags.NoClip;
```
This flag directs `DrawString` not to clip text that lies partially outside the display rectangle. The result is that two full lines of text are displayed for every enumeration value except `StringTrimming.None`.

For the `StringTrimming.None` case, the entire block of text is now displayed. The combination of this enumeration value and the `NoClip` format flag essentially negates the effect of the bottom of the rectangle.

If you include both flags, like so,
```
strfmt.FormatFlags |= StringFormatFlags.NoClip;
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```
then for all `StringTrimming` values except `None`, the effect is the same as with just the `NoWrap` flag. For `StringTrimming.None`, the text is not wrapped and also not prevented from going past the right margin. It's as if you specified a `PointF` rather than a `RectangleF` in the `DrawString` call.

When you're displaying text in a rectangle, you need to watch out for clipping. If you make the rectangle height an integral multiple of the line-spacing value, you won't have a problem with clipping. That's probably the best solution. Otherwise, you should set the `NoClip` format flag to prevent clipping. But keep in mind that doing so will possibly cause the last line of text to be partially displayed beyond the bottom of the rectangle. In some cases (if the height of the rectangle is the height of your client area, for example), the last line of text will be clipped anyway because it exceeds the boundary of the client area. Be sure to adjust the rectangle so that all lines of text are displayed.

If you create a `StringFormat` object using the default constructor
```
StringFormat strfmt = new StringFormat();
```
or if you create it using this static property of `StringFormat`
```
StringFormat strfmt = StringFormat.GenericDefault;
```
the `Trimming` property is initially set to `StringTrimming.Character`. If you create the `StringFormat` object using the static property
```
StringFormat strfmt = StringFormat.GenericTypographic;
```
the `Trimming` property is initially set to `StringTrimming.None` and the `NoClip` format flag is set.

**Start a Tab**

Tab stops govern how the `DrawString` call interprets the Unicode tab character ‘\t’ or ‘\x0009’. If your `DrawString` call doesn't include a `StringFormat` argument, the default tab stops, measured in points, are equal to four times the size of the font. (In other words, the tabs are four ems.) For example, a 9-
point font will have tab stops every 36 points, or 1/2 inch; an 18-point font will have tab stops every 72 points, or 1 inch; and a 36-point font will have tab stops every 144 points, or 2 inches. The tab stops are measured from where the text begins as indicated by the PointF or RectangleF argument to the DrawString call.

If your DrawString call includes a StringFormat argument, default tab stops don't exist and the DrawString method ignores all tab characters in the text. You need to set tab stops using the SetTabStops method of StringFormat. The StringFormat class also includes a method to obtain the current tab stop settings:

```
StringFormat Methods (selection)

void SetTabStops(float fFirstTab, float[] afTabs)
float[] GetTabStops(out float fFirstTab)
```

Tab stops are in world coordinates. You'll notice that the tab stops are specified by both a float value and an array of float values, which would seem to indicate that the value not in the array is treated differently. What the method syntax doesn't imply is that the last value of the array is also treated differently.

Let me give you a few simple examples first before I show you how the SetTabStops method works in the general case. I'll assume the page units are set to GraphicsUnit.Point.

If you need just one tab stop, say at 4 inches (288 points), you can specify that as the first argument to the method and make the array contain just a single 0:

```csharp
strfmt.SetTabStops(288, new float[] { 0 });
```

You can't set the array argument to null. You can also use

```csharp
strfmt.SetTabStops(0, new float[] { 288, 0 });
```

If you need two tabs stops, for example, at 1 inch (72 points) and 3 inches (216 points), you use

```csharp
strfmt.SetTabStops(0, new float[] { 72, 144, 0 });
```

Notice that the second array element is the difference between 72 and 216 points. Although I'm showing the array being created directly in the SetTabStops call, you can, of course, define it outside the method.

If you need tab stops every 0.5 inch (36 points), you use

```csharp
strfmt.SetTabStops(0, new float[] { 36 });
```

The tab stops will be at 36 points, 72 points, 108 points, 144 points, and so forth.

As you can see, the SetTabStops method can define both discrete tab stops and repeating tab stops, and it's the combination of these two that makes this method so complicated. In the general case, the arguments to SetTabStops look like this:

```csharp
strfmt.SetTabStops(S, new float[] { A, B, C, ..., N, R });
```

I'm using the letter R to stand for Repeating and the letter S to indicate Shift. Any of these values can be 0 or negative. The SetTabStops method sets tab stops at the following positions measured from the starting position of the text:

```
S + A
S + A + B
S + A + B + C
S + A + B + C + ... + N
S + A + B + C + ... + N + R
```

In addition, the method also sets tab stops at positions R, 2R, 3R, and so forth, but these repeating tab stops begin only after the longest of the other tab stops. For example, the call
_authentication.SetTabStops(100, new float[] { 50, 75, 50, 100 });

sets tab stops at 150, 225, 275, 375, 400, 500, 600, and so on. Units are world coordinates.

You can set the last element of the array (which I’ve called R) to 0 if you want all the tab stops to be explicitly defined. You can also set S to 0. However, it’s possible to use that initial argument to SetTabStops intelligently. For example, you could first define an array that has four tab stops measured from a horizontal coordinate of 0:

float[] afTabs = { 100, 150, 100, 50, 0 };

Notice that the last argument is 0, so there will be no repeating tab stops.

If you’re preparing to display text starting at a horizontal coordinate of 0, you can call SetTabStops with an initial argument of 0:

_strfmt.SetTabStops(0, afTabs);

This call sets tabs at 100, 250, 350, and 400 units. However, you might now need to display text starting at a horizontal coordinate of 50, but you want the tab stops in the same physical locations. You can do that by passing –50 as the first argument to SetTabStops:

_strfmt.SetTabStops(-50, afTabs);

Now the tab stops are 50, 200, 300, and 350 but measured from the starting coordinate of 50, so they’re really 100, 250, 350, and 400, the same as before.

Let’s put a lot of what we’ve learned here into practice by formatting a chunk of text into columns. The text I’m using is the beginning of Edith Wharton’s 1920 novel The Age of Innocence. The following class has a single read-only Text property that returns the first five paragraphs of the novel.

**AgeOfInnocence.cs**

```csharp
//--------------------------------------------------------------------
// AgeOfInnocence.cs © 2001 by Charles Petzold; text by Edith Wharton
//--------------------------------------------------------------------
class AgeOfInnocence
{
    public static string Text
    {
        get
        {
            return "On a January evening of the early seventies, Christine Nilsson was " +
            "singing in Faust at the Academy of Music in New York." +
            "\n" +
            "Though there was already talk of the erection, in remote metropolitan " +
            "distances \"above the Forties,\" of a new Opera House which should " +
            "compete in costliness and splendour with those of the great European " +
            "capitals, the world of fashion was still content to reassemble every " +
            "winter in the shabby red and gold boxes of the sociable old Academy. " +
            "Conservatives cherished it for being small and inconvenient, and thus " +
            "keeping out the \"new people\" whom New York was beginning to dread and " +
            "yet be drawn to; and the sentimental clung to it for its historic " +
            "associations, and the musical for its excellent acoustics, always so " +
```
"problematic a quality in halls built for the hearing of music." +
"\n" +
"It was Madame Nilsson's first appearance that winter, and what the " +
daily press had already learned to describe as "an exceptionally " +
brilliant audience" had gathered to hear her, transported through the " +
slippery, snowy streets in private broughams, in the spacious family " +
landau, or in the humbler but more convenient "Brown &c&o&u&p&é." To " +
come to the Opera in a Brown &c&o&u&p&é was almost as honourable a way " +
of arriving as in one's own carriage; and departure by the same means " +
had the immense advantage of enabling one (with a playful allusion to " +
democratic principles) to scramble into the first Brown conveyance in " +
the line, instead of waiting till the cold-and-gin congested nose of " +
one's own coachman gleamed under the portico of the Academy. It was one " +
of the great livery-stableman's most masterly intuitions to have " +
discovered that Americans want to get away from amusement even more " +
quickly than they want to get to it." +
"\n" +
"When Newland Archer opened the door at the back of the club box the " +
curtain had just gone up on the garden scene. There was no reason why " +
the young man should not have come earlier, for he had dined at seven, " +
alone with his mother and sister, and had lingered afterward over a " +
cigar in the Gothic library with glazed black-walnut bookcases and " +
finial-topped chairs which was the only room in the house where Mrs. " +
Archer allowed smoking. But, in the first place, New York was a " +
metropolis, and perfectly aware that in metropolises it was "not the " +
thing" to arrive early at the opera; and what was or was not "the " +
thing" played a part as important in Newland Archer's New York as the " +
inscrutable totem terrors that had ruled the destinies of his " +
forefathers thousands of years ago." +
"\n" +
"The second reason for his delay was a personal one. He had dawdled " +
over his cigar because he was at heart a dilettante, and thinking over a " +
pleasure to come often gave him a subtler satisfaction than its " +
realisation. This was especially the case when the pleasure was a " +
delicate one, as his pleasures mostly were; and on this occasion the " +
moment he looked forward to was so rare and exquisite in quality " +
that\`x2014well, if he had timed his arrival in accord with the prima " +
donna's stage-manager he could not have entered the Academy at a more " +
significant moment than just as she was singing: "He loves me\`x2014he " +

"loves me not" and sprinkling the falling "+
"daisy petals with notes as clear as dew." +
"\n";
}
}
}
}
Notice that I've used a tab character to indent the first line of every paragraph except the first. The
text includes a few italicized words; I used the ampersand technique I discussed earlier to make
these words underlined instead. The text has some em dashes as well.

Here's the program that formats this text into columns. Each column requires a DrawString call. I
originally wrote this program to set a page transform that lets me express units in terms of picas. A
pica is equal to 12 points, or (in computer typography) 1/6 inch. I wanted to use picas because the
widths of columns in magazines and newspapers are generally measured in picas. I decided to make
my columns 12 picas wide (that is, 2 inches) with a 1-pica space between the columns. However, at
the time this was written, converting to picas caused a problem in the DrawString method, so I've
changed the program to use units of points.

TextColumns.cs
//------------------------------------------
// TextColumns.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
class TextColumns: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TextColumns());
    }
    public TextColumns()
    {
        Text = "Edith Wharton's \"The Age of Innocence\";"
        Font = new Font("Times New Roman", 10);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        int iChars, iLines;
        string str = AgeOfInnocence.Text;
        StringFormat strfmt = new StringFormat();

        // Set units of points while converting dimensions.
PointF[] aptf = { new PointF(cx, cy) };
grfx.TransformPoints(CoordinateSpace.Device,
                        CoordinateSpace.Page, aptf);
grfx.PageUnit  = GraphicsUnit.Point;
grfx.TransformPoints(CoordinateSpace.Page,
                        CoordinateSpace.Device, aptf);
float fcx = aptf[0].X;
float fcy = aptf[0].Y;

    // StringFormat properties, flags, and tabs
    strfmt.HotkeyPrefix = HotkeyPrefix.Show;
    strfmt.Trimming     = StringTrimming.Word;
    strfmt.FormatFlags |= StringFormatFlags.NoClip;
    strfmt.SetTabStops(0, new float[] { 18 });

    // Display text.
    for (int x = 0; x < fcx && str.Length > 0; x += 156)
    {
        RectangleF rectf = new RectangleF(x, 0, 144,
                                           fcy -
                                           Font.GetHeight(grfx));

        grfx.DrawString(str, Font, brush, rectf, strfmt);
        grfx.MeasureString(str, Font, rectf.Size, strfmt,
                            out iChars, out iLines);

        str = str.Substring(iChars);
    }
On a January evening of the early nineteen hundred, Christine Nilsson was singing in Faust at the Academy of Music in New York.

Though there was already talk of the erection of another metropolitan opera house—above the Biltmore—a new Opera House which should compete in comfort and splendor with those of the great European capitals, the world of fashion was still content to reassure itself. Writer in the shabby red and gold boxes of the sensible old Academy, Conservator, cherished it for being small and inconvenient, and thus keeping out the "new people" whom New York was beginning to draw, yet being known to, and the sentimental bond to it for its historic associations, and the musica, for its excellent acoustics, always reproachable 

It was Madame Nilsson's appearance that saved, and the
the daily press had already learned to describe as "an exceptionally brilliant soloist" who was to appear in New York, transported through the city's money streets in private broughams, undaunted by the vast family audience, and in the hands of her music, yet as unostentatious and devoted as Browning, "To come to the Opera..."
Chapter 10: The Timer and Time

Overview

The timer is an input device that periodically notifies an application when a specified interval of time has elapsed. Your program defines the interval, in effect saying, "Give me a nudge every 10th second." The timer then triggers an event handler in your program 10 times a second.

Three different classes defined in the System.Timers, System.Threading, and System.Windows.Forms namespaces are named Timer. I'll be using the one defined in System.Windows.Forms, which is the timer that Microsoft Windows programmers are familiar with. It is integrated with the other Windows events and involves the lowest hassle factor.

While obviously not as important an input device as the keyboard and the mouse, the timer is surprisingly useful and finds its way into many Windows Forms applications. The obvious timer application is a clock, and indeed, this chapter is overflowing with clock applications. But here are some other uses for the timer, some perhaps not so obvious:

- **Multitasking** Although Windows is a preemptive multitasking environment, usually it's advisable for a program to return control to Windows as quickly as possible after processing an event. Not doing so tends to gum up the works. If a program must do a large amount of processing, it can divide the job into smaller pieces and process each piece upon receipt of a timer event.

- **Maintaining an updated status report** A program can use the timer to display real-time updates of continuously changing information, such as a display of resources or the progress of certain tasks.

- **Implementing an "autosave" feature** The timer can prompt an application to save a user's work to disk whenever a specified period of time has elapsed.

- **Terminating demo versions of programs** Some demonstration versions of programs are designed to terminate, say, 30 minutes after they begin. The timer can signal such applications when the time is up. (An example is the CloseInFive program coming up soon in this chapter.)

- **Pacing movement** Graphical objects in a game or successive displays in a computer-assisted instruction program usually need to proceed at a set rate. Using the timer eliminates the inconsistencies that might result from variations in microprocessor speed. Animation often makes use of the timer.

You can also think of the timer as a guarantee that a program can regain control sometime in the future. Whenever a program relinquishes control after executing the code in the constructor or an event handler, it usually can't determine when the next event will occur. The timer is more certain.

I say *more* certain because the timer doesn't have the rhythmical consistency of a metronome. The events triggered by the Timer class are synchronous with the other events. In other words, a timer event will never interrupt the processing of another event in the same execution thread. Code that spends a long time processing an event will delay a timer event.

The Timer Class

The Timer class is small and relatively simple. You generally create a Timer object using the default constructor:

```csharp
Timer timer = new Timer();
```

The Timer has one event:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tick</td>
<td>OnTick</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

Somewhere in your class you'll have an event handler for the timer defined like so:

```csharp
void TimerOnTick(object obj, EventArgs ea)
{
    //
}
```
You can name it whatever you want, of course. You attach this event handler to the Timer object you've created:

timer.Tick += new EventHandler(TimerOnTick);

The Timer class has just two properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Interval</td>
<td>get/set</td>
<td>Tick time in milliseconds</td>
</tr>
<tr>
<td>bool</td>
<td>Enabled</td>
<td>get/set</td>
<td>Set to true if timer is running</td>
</tr>
</tbody>
</table>

You set the Interval property to the number of milliseconds between calls to your event handler. For example, this statement sets the tick time to 1 second:

timer.Interval = 1000;

Although you can set the Interval property to values as low as 1, you're not guaranteed to get a tick time of 1 millisecond. Under Windows 2000, for example, even if you set an Interval of 1, the timer will call your event handler approximately every 10 milliseconds.

Windows rounds intervals you specify to the next highest multiple of the period of the operating system's internal clock. Under Windows 2000, for example, specifying an Interval of 11 through 20 results in an actual interval of 20 milliseconds. But an Interval of 20 doesn't imply that you'll get precisely 50 calls every second. If a timer event is delayed more than 20 milliseconds, it's consolidated with the next timer event. There are never multiple pending timer events.

But you won't get any calls to your event handler unless you also enable the timer:

timer.Enabled = true;

Alternatively, you can use these two methods:

```csharp
Timer Methods (selection)

void Start()
void Stop()
```

The Start call is equivalent to setting Enabled to true, and the Stop call is equivalent to setting Enabled to false.

You can change these properties in your timer event handler, effectively resetting the timer. Remember that the first argument to the event handler is the object associated with the event handler, which in this case is the Timer object, so you can cast it like so:

```csharp
Timer timer = (Timer) obj;
```

Here's a program that sets a one-shot timer, so called because the timer event handler turns the timer off.

```csharp
CloseInFive.cs

// CloseInFive.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;
```
class CloseInFive: Form
{
    public static void Main()
    {
        Application.Run(new CloseInFive());
    }

    public CloseInFive()
    {
        Text = "Closing in Five Minutes";

        Timer timer = new Timer();
        timer.Interval = 5 * 60 * 1000;
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Enabled = true;
    }

    void TimerOnTick(object obj, EventArgs ea)
    {
        Timer timer = (Timer) obj;

        timer.Stop();
        timer.Tick -= new EventHandler(TimerOnTick);

        Close();
    }
}

Not only does the timer event handler turn the timer off, but it also closes the program. This program is an example of how you would implement a demo feature that allows the user to experience the benefits of an application but not actually use it much.

When closing a program, it's not necessary to stop the timer and detach the event handler. However, if you truly are using a timer for a one-shot operation, it's a good idea to do so: the Timer object will then qualify for garbage collection.

At the other extreme from CloseInFive, here's a hypnotic program that sets the timer once and runs forever.

RandomRectangle.cs
//-----------------------------
// RandomRectangle.cs © 2001 by Charles Petzold
//-----------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class RandomRectangle: Form
{
    public static void Main()
    {
        Random random = new Random();
        int x = random.Next(500);
        int y = random.Next(500);
        RandomRectangle rect = new RandomRectangle()
        {
            Text = "Random Rectangle";
            Top = y;
            Left = x;
            Width = random.Next(100);
            Height = random.Next(100);
        }

        Application.Run(rect);
    }
}
Application.Run(new RandomRectangle());
}
public RandomRectangle()
{
    Text = "Random Rectangle";
    Timer timer = new Timer();
timer.Interval = 1;
timer.Tick += new EventHandler(TimerOnTick);
timer.Start();
}
void TimerOnTick(object obj, EventArgs ea)
{
    Random rand = new Random();
    int x1 = rand.Next(ClientSize.Width);
    int x2 = rand.Next(ClientSize.Width);
    int y1 = rand.Next(ClientSize.Height);
    int y2 = rand.Next(ClientSize.Height);
    Color color = Color.FromArgb(rand.Next(256),
                                  rand.Next(256),
                                  rand.Next(256));

    Graphics grfx = CreateGraphics();
grfx.FillRectangle(new SolidBrush(color),
                            Math.Min(x1, x2), Math.Min(y1, y2),
                            Math.Abs(x2 - x1), Math.Abs(y2 - y1));
grfx.Dispose();
}

The **DateTime** Structure

If you want to write a clock application, you need to know something about the representation of date
and time in the .NET Framework.

The most important date and time structure, appropriately named *DateTime*, is defined in the *System*
namespace. You can create an object of type *DateTime* using one of its seven constructors, three of
which are listed here:

**DateTime Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime(int year, int month, int day)</td>
</tr>
<tr>
<td>DateTime(int year, int month, int day, int hour, int minute, int second)</td>
</tr>
<tr>
<td>DateTime(int year, int month, int day, int hour, int minute, int second, int msec)</td>
</tr>
</tbody>
</table>
The year can range from 1 to 9999, the month can range from 1 to 12, the day can range from 1 through the number of days in that month and year, the hour can range from 0 through 23, the minute and second can range from 0 through 59, and the milliseconds argument can range from 0 through 999. If any of the arguments is out of range, the constructor throws an exception.

The `DateTime` constructor also throws an exception if the combination of year, month, and day arguments isn't consistent. For example, a month of 2 and a day of 29 is acceptable only for a leap year. These `DateTime` constructors use leap year rules associated with the Gregorian calendar (which was instituted by Pope Gregory XIII in 1582 and eventually adapted worldwide in the years and centuries that followed). In the Gregorian calendar, a year is a leap year if it is divisible by 4 but not divisible by 100 unless it is divisible by 400. The year 1900 is not a leap year; 2000 is. (Prior to the Gregorian calendar, leap years were celebrated every four years without exception.) The `DateTime` constructor observes these same leap year rules even for years preceding the invention of the Gregorian calendar.

`DateTime` has 15 properties, all of which are read-only, and 10 of which are shown here:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Year</td>
<td>get</td>
<td>1 through 9999</td>
</tr>
<tr>
<td>int</td>
<td>Month</td>
<td>get</td>
<td>1 through 12</td>
</tr>
<tr>
<td>int</td>
<td>Day</td>
<td>get</td>
<td>1 through 31</td>
</tr>
<tr>
<td>int</td>
<td>Hour</td>
<td>get</td>
<td>0 through 23</td>
</tr>
<tr>
<td>int</td>
<td>Minute</td>
<td>get</td>
<td>0 through 59</td>
</tr>
<tr>
<td>int</td>
<td>Second</td>
<td>get</td>
<td>0 through 59</td>
</tr>
<tr>
<td>int</td>
<td>Millisecond</td>
<td>get</td>
<td>0 through 999</td>
</tr>
<tr>
<td>int</td>
<td>DayOfWeek</td>
<td>get</td>
<td>0 (Sunday) through 6 (Saturday)</td>
</tr>
<tr>
<td>int</td>
<td>DayOfYear</td>
<td>get</td>
<td>1 through 366</td>
</tr>
<tr>
<td>DateTime</td>
<td>Date</td>
<td>get</td>
<td>Time set to 0 (midnight)</td>
</tr>
</tbody>
</table>

The first seven properties are the familiar components of the date and time, and will match the values set in the constructor. The `DayOfWeek` and `DayOfYear` properties provide some additional information about the date. The `Date` property returns a `DateTime` object that represents the same day as the current `DateTime` object but with the `Hour`, `Minute`, `Second`, and `Millisecond` properties set to 0.

`DateTime` has three static properties, which are particularly useful:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime</td>
<td>Now</td>
<td>get</td>
<td>Current local date and time</td>
</tr>
<tr>
<td>DateTime</td>
<td>Today</td>
<td>get</td>
<td>Current local date</td>
</tr>
<tr>
<td>DateTime</td>
<td>UtcNow</td>
<td>get</td>
<td>Current UTC date and time</td>
</tr>
</tbody>
</table>

The `DateTime.Now` property returns a `DateTime` structure filled in with the current local date and time. For example, to obtain the current date and time in your program, call

```csharp
DateTime dt = DateTime.Now;
```

Now you can use the `DateTime` properties in `dt` to obtain the components of the current date and time. The `DateTime.Today` property is similar but returns a `DateTime` structure with today's date and all the time components set to 0.

The static `UtcNow` property returns a `DateTime` structure with the current date and time in Coordinated Universal Time (UTC), which I'll discuss in the next section.
The DateTime structure contains a number of methods and overloaded operators that let you perform calculations on dates and times. The comparison operators (==, !=, <, >, <=, and >=) are all valid for DateTime objects. Addition and subtraction are also supported, but these operations involve TimeSpan objects that I'll get to shortly.

**Local Time and Universal Time**

People everywhere around the world like to think of noon as the time when the sun is highest in the sky and midnight as the middle of the night. Because these two events don't occur everywhere on the earth at the same time, people living in different areas of the world set their clocks differently. Once a chaotic practice, this tendency has evolved into strict time zones generally set by national governments and calculated as hour or half-hour offsets from Greenwich Mean Time.

Greenwich, England, has played an important role in the evolution of time standards because it is the site of the Royal Greenwich Observatory (RGO). The RGO was founded in 1675 to develop techniques of astronomical navigation for ships at sea. In the 1760s, the observatory began publishing nautical almanacs that for convenience placed the prime meridian (the line of 0º longitude) at Greenwich. This system of meridians was eventually agreed upon as a world standard in 1884, although the French continued to use Paris as the prime meridian until 1911.

Earlier, in 1833, Greenwich astronomers began dropping a ball that was visible to ships in the Thames every day at 1:00 p.m. That was the origin of Greenwich Mean Time. In the 1840s, Greenwich Mean Time was declared the standard time for all of Great Britain to replace various local times that had developed over the years.

While people often still refer to Greenwich Mean Time as the world standard, the use of the term Coordinated Universal Time (UTC) is considered more scientifically correct. (Coordinated Universal Time is abbreviated UTC as something of a compromise between the English word order—which would imply the abbreviation CUT—and the French Temps Universel Coordonné, which has the abbreviation TUC.) By international agreement since 1972, UTC is the same all over the world.

Local standard time is a positive or negative offset from UTC. Time zones to the west of Greenwich are behind UTC, and time zones to the east of Greenwich are ahead of UTC. For example, Eastern Standard Time, which includes the east coast of the United States, is UTC minus 5 hours. This is not the same as UTC plus 19 hours. Such a calculation results in the correct time but the incorrect day.

Then there's that quaint custom known as daylight saving time. The principle behind it is simple: as the summer solstice approaches, the sun is rising earlier and setting later, so it's no big deal to get out of bed a little earlier and enjoy even more sun in the evening. Some countries observe daylight saving time and some don't, and those that observe it frequently begin and end it on different dates. Even within some countries, notably the United States, daylight saving time is implemented inconsistently—many states observe it, but some don't.

In Windows, you can set the time zone for your machine using the Date/Time Properties dialog box that you can open from Control Panel or by double-clicking the time in the Windows taskbar. You can also indicate whether the system should automatically adjust for daylight saving time. Obviously, for a particular machine, the local time is just an offset of the UTC based on both the local time zone and the effect of daylight saving time.

The DateTime structure by itself doesn't imply UTC or a local time. When you use one of the DateTime constructors, you are specifying a date and time that may be UTC, a local time, the date and time of your birth, or anything you want.

As I mentioned earlier, the properties Now and Today return local date and time, and UtcNow returns UTC based on the time zone settings of the current machine. You can convert the time stored in a DateTime object between local time and UTC by using the following methods:

**DateTime Methods (selection)**

- `DateTimeToLocalTime()`
- `DateTimeToUniversalTime()`
For example, if the variable `dtLocal` contains a local date and time, you can convert it to a UTC date and time by calling

```csharp
DateTime dtUtc = dtLocal.ToUniversalTime();
```

If `dtLocal` actually contained a time in UTC, what you end up with here is nonsense. It's your responsibility to keep track of what time zones your `DateTime` objects pertain to.

You can get information about the time zone defined for the current machine, and the daylight saving time rules associated with that time zone, from the `TimeZone` class, also defined in the `System` namespace. `TimeZone` is defined as abstract, which means that you can't instantiate it; you can only instantiate a subclass of `TimeZone`. However, the class by itself provides some useful information. `TimeZone` has one static property, which returns an instance of the class. This instance represents the time zone that is set for the current machine:

<table>
<thead>
<tr>
<th>TimeZone Static Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><code>TimeZone</code></td>
</tr>
</tbody>
</table>

For example, if you call

```csharp
TimeZone tz = TimeZone.CurrentTimeZone;
```

tz represents the time zone set on the current computer. The two nonstatic properties provide names associated with the time zone:

<table>
<thead>
<tr>
<th>TimeZone Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><code>string</code></td>
</tr>
<tr>
<td><code>string</code></td>
</tr>
</tbody>
</table>

For example, if your machine is located on the east coast of the United States, the time zone is set correctly, and you've set the tz variable as shown previously, the `tz.StandardName` property returns “Eastern Standard Time” and `tz.DaylightName` returns “Eastern Daylight Time.”

Here are the `TimeZone` methods:

<table>
<thead>
<tr>
<th>TimeZone Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TimeSpan GetUtcOffset(DateTime dt)</code></td>
</tr>
<tr>
<td><code>DateTime ToLocalTime(DateTime dt)</code></td>
</tr>
<tr>
<td><code>DateTime ToUniversalTime(DateTime dt)</code></td>
</tr>
<tr>
<td><code>DaylightTime GetDaylightChanges(int iYear)</code></td>
</tr>
<tr>
<td><code>bool IsDaylightSavingTime(DateTime dt)</code></td>
</tr>
<tr>
<td><code>bool IsDaylightSavingTime(DateTime dt, DaylightTime dlt)</code></td>
</tr>
</tbody>
</table>

The `GetUtcOffset` method returns an offset between the time in the particular time zone and UTC expressed as a `TimeSpan` object. (I'll discuss `TimeSpan` in more detail shortly. For now, be aware that it's a structure defined in the `System` namespace that's used to express durations of time in units of 100 nanoseconds.) `GetUtcOffset` takes daylight saving time into account. Using the same example of the computer on the east coast of the United States, the method

```csharp
tz.GetUtcOffset(new DateTime(2002, 2, 2))
```

returns −5:00:00, which signifies −5 hours. That's what you add to UTC to get Eastern Standard Time. The method call

```csharp
tz.GetUtcOffset(new DateTime(2002, 8, 29))
```
returns –4:00:00, or –4 hours. That’s the effect of daylight saving time.

The ToLocalTime and ToUniversalTime methods are similar to the methods in the DateTime structure but are based on the particular TimeZone object. If you had a way to get TimeZone objects for other time zones around the world, you could use these methods to convert between local and UTC for other time zones.

The statement

```csharp
DaylightTime dlt = tz.GetDaylightChanges(2002);
```

returns an object of type DaylightTime, a class defined in the System.Globalization namespace. It has three properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime</td>
<td>Start</td>
<td>get</td>
</tr>
<tr>
<td>DateTime</td>
<td>End</td>
<td>get</td>
</tr>
<tr>
<td>TimeSpan</td>
<td>Delta</td>
<td>get</td>
</tr>
</tbody>
</table>

The Start and End properties indicate that daylight saving time begins on April 7, 2002, at 2:00 a.m., and ends on October 27, 2002, at 2:00 a.m. (Both transitions occur on Sunday mornings.) The Delta value is the time difference, which is 1 hour.

For time zones in the northern hemisphere, Start is earlier in the year than End. For time zones in the southern hemisphere, the seasons are switched: daylight saving time starts later in the year and ends early in the next year.

The Tick Count

Another way of representing date and time is by a number of 100-nanosecond clock ticks. Internally, the DateTime structure stores the date and time as the number of ticks since midnight, January 1, 1 C.E. The two remaining DateTime properties provide that value as well as the number of ticks since midnight:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Ticks</td>
<td>get</td>
<td>100-nanosecond intervals since 1/1/0001</td>
</tr>
<tr>
<td>TimeSpan</td>
<td>TimeOfDay</td>
<td>get</td>
<td>Ticks since midnight</td>
</tr>
</tbody>
</table>

Remember that a long is 64 bits wide and hence is adequate for storing the very large numbers involved here. Notice that the TimeOfDay property returns an object of type TimeSpan, which expresses durations of time in units of 100 nanoseconds.

For the date January 1, 2001, the Ticks property returns the value 631,139,040,000,000,000. There are 10,000 ticks in a millisecond, 10,000,000 ticks in a second, 600,000,000 ticks in a minute, 36,000,000,000 ticks in an hour, and 864,000,000,000 ticks in a day. (All these constant values are available as fields in the TimeSpan structure, by the way.) That means that 730,485 days have elapsed in those 2000 years, for an average of 365.2425 days per year.

The value of 365.2425 days per year is correct for the Gregorian calendar: most years have 365 days. An extra day every four years adds 0.25 to the average days per year. Excluding an extra day every 100 years lessens the average days per year by 0.01. Including an extra day every 400 years increases the average days per year by 0.0025. In other words,

\[365 + \frac{1}{4} - \frac{1}{100} + \frac{1}{400} = 365.2425\]

You can create a DateTime object from a long value indicating the number of ticks since January 1, 1 C.E.:
You can create `TimeSpan` objects similarly, and you can also create `TimeSpan` objects with explicit day and time values. Here's a complete list of the `TimeSpan` constructors:

**TimeSpan Constructors**

- `TimeSpan(long ticks)`
- `TimeSpan(int hours, int minutes, int seconds)`
- `TimeSpan(int days, int hours, int minutes, int seconds)`
- `TimeSpan(int days, int hours, int minutes, int seconds, int milliseconds)`

The `DateTime` and `TimeSpan` constructors look very similar, so it's important to understand the distinction between the two structures. A `DateTime` object represents a particular date and time; the `Ticks` property of `DateTime` is the number of 100-nanosecond intervals since January 1, 1 C.E.

A `TimeSpan` object represents a duration—a period of elapsed time. Notice that the arguments of the `TimeSpan` constructors don't represent any particular day, hour, minute, second, and millisecond. They represent a certain number of days, and a number of hours, and so forth. There are no `TimeSpan` constructors that involve months and years because months and years don't have a fixed number of days.

The values that can be assigned to arguments in the `TimeSpan` constructors aren't limited, unlike those of `DateTime` constructors. For example, the statement

```csharp
TimeSpan ts = new TimeSpan(1000, 1000, 1000, 1000, 1000);
```

is perfectly legal.

Here's a complete list of the `TimeSpan` properties:

**TimeSpan Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long</code></td>
<td><code>Ticks</code></td>
<td>get</td>
<td>Number of 100-nanosecond intervals</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Days</code></td>
<td>get</td>
<td>Whole number of days</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Hours</code></td>
<td>get</td>
<td>0 through 23</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Minutes</code></td>
<td>get</td>
<td>0 through 59</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Seconds</code></td>
<td>get</td>
<td>0 through 59</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Milliseconds</code></td>
<td>get</td>
<td>0 through 999</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>TotalDays</code></td>
<td>get</td>
<td>= (double) Ticks / TicksPerDay</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>TotalHours</code></td>
<td>get</td>
<td>= (double) Ticks / TicksPerHour</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>TotalMinutes</code></td>
<td>get</td>
<td>= (double) Ticks / TicksPerMinute</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>TotalSeconds</code></td>
<td>get</td>
<td>= (double) Ticks / TicksPerSecond</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>TotalMilliseconds</code></td>
<td>get</td>
<td>= (double) Ticks / TicksPerMillisecond</td>
</tr>
</tbody>
</table>

For the last five properties, I've indicated how they are calculated from convenient fields in the `TimeSpan` structure.
I mentioned earlier that the `DateTime` structure defines an overload of the addition operator. However, you can add only a `TimeSpan` object to a `DateTime` object. If `dt`, `dt1`, and `dt2` are `DateTime` objects and `ts` is a `TimeSpan` object, you can perform addition like this:

```
    dt2 = dt1 + ts;
```

or like this:

```
    dt += ts;
```

For example, you may want to increase a time and date by 45 minutes:

```
    dt += new TimeSpan(0, 45, 0);
```

or 1 week:

```
    dt += new TimeSpan(7, 0, 0, 0);
```

This is the safe way to perform these calculations. (Another safe approach is to use the various `Add` methods of the `DateTime` class.) The subtraction operator is defined in two ways. You can subtract one date and time from another to get a `TimeSpan` object:

```
    ts = dt2 - dt1;
```

Or you can subtract a `TimeSpan` object from a `DateTime` object to get another `DateTime` object:

```
    dt2 = dt1 - ts;
```

or

```
    dt -= ts;
```

`TimeSpan` objects can also be added, subtracted, or compared to each other in any way.

### Calendars Around the World

Here are the final three constructors for `DateTime`:

**DateTime Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime(int year, int month, int day, Calendar cal)</td>
</tr>
<tr>
<td>DateTime(int year, int month, int day, int hour, int minute, int sec,</td>
</tr>
<tr>
<td>Calendar cal)</td>
</tr>
<tr>
<td>DateTime(int year, int month, int day, int hour, int minute, int sec,</td>
</tr>
<tr>
<td>int msec, Calendar cal)</td>
</tr>
</tbody>
</table>

The final argument is an object of type `Calendar`. This argument indicates how the year, month, and day arguments are to be interpreted. As I mentioned earlier, the constructors without the `Calendar` argument are assumed to refer to dates in the Gregorian calendar.

`Calendar` is an abstract class defined in the `System.Globalization` namespace, a namespace that also includes eight classes derived from `Calendar`.
Hijri is another name for the Islamic calendar.

When you include a Calendar object as the last argument to the constructor, different consistency rules are applicable. For example,

```csharp
DateTime dt = new DateTime(1900, 2, 29);
```

generates an exception because 1900 isn't a leap year in the Gregorian calendar. However,

```csharp
DateTime dt = new DateTime(1900, 2, 29, new JulianCalendar());
```

doesn't cause an exception because in the Julian calendar every year divisible by 4 is a leap year.

Moreover, if you actually make that call using the JulianCalendar object and then look at the individual properties of the DateTime structure, you'll find that Month equals 3 (March) and Day equals 13. The Year, Month, and Day properties of the DateTime structure always represent dates in the Gregorian calendar. The constructor converts a date in a particular calendar into a tick count; the DateTime properties convert from the tick count to dates in the Gregorian calendar.

The original adoption of the Gregorian calendar caused the date after October 4, 1582, to be October 15, 1582, effectively skipping 10 days. \[1\] If you call

```csharp
dt = new DateTime(1582, 10, 5, new JulianCalendar());
```

the Month property will be 10 and the Day property will indeed be 15.

It gets more interesting. Suppose you call
dt = new DateTime(5762, 5, 20, new HebrewCalendar());

Yes, that is indeed a year in the Hebrew calendar—the 20th day in the month of Shevat in the year 5762. The resultant \texttt{DateTime} structure has a \texttt{Year} property of 2002, and \texttt{Month} and \texttt{Day} properties both equal to 2. Basically, what you have here is a conversion between the Hebrew calendar and the Gregorian calendar. When the last argument to the \texttt{DateTime} constructor is a \texttt{HebrewCalendar} object, the \texttt{Month} argument can be set to 13 in some years.

Similarly, you can specify a date in the Islamic calendar:

dt = new DateTime(1422, 11, 20, new HijriCalendar());

That's the 20th day of the month of Dhu'l-Qa'dah in the year 1422. Again, the resultant \texttt{DateTime} structure has a \texttt{Year} property of 2002, and \texttt{Month} and \texttt{Day} properties both equal to 2.

To convert from a Gregorian date to another calendar, you need to create instances of the particular calendar, for example,

\begin{verbatim}
HebrewCalendar hebrewcal = new HebrewCalendar();
HijriCalendar hijrical = new HijriCalendar();
\end{verbatim}

You also need a \texttt{DateTime} object:

\begin{verbatim}
DateTime dt = new DateTime(2002, 2, 2);
\end{verbatim}

To convert this Gregorian date into a date in the Hebrew or Islamic calendar, you use the following three methods:

\begin{verbatim}
Calendar Methods (selection)
int GetYear(DateTime dt)
int GetMonth(DateTime dt)
int GetDayOfMonth(DateTime dt)
\end{verbatim}

For example, the expression

\begin{verbatim}
hijrical.GetYear(dt)
\end{verbatim}

returns 1422.

The years leading up to the recent millennium anniversary saw the publication of several books that retold the history of the Julian and Gregorian calendars. Perhaps the shortest and most eloquent is Stephen Jay Gould, \textit{Questioning the Millennium: A Rationalist's Guide to a Precisely Arbitrary Countdown} (New York: Harmony Books, 1997).

\textbf{A Readable Rendition}

Some of the most important methods in \texttt{DateTime} are those that format the date and time into human-readable form. This conversion might seem fairly trivial until you realize that people all over the world write dates and times in different ways. Some cultures use 24-hour clocks; others prefer using a.m. and p.m. Some cultures write the day before the month; others put the month before the day. If the date includes month names or days of the week, it's helpful for these names to be in the user's language. Even within a particular culture, some users may have individual preferences that differ from the cultural standards.

These cultural standards and user preferences are accessible through the Regional Options dialog box in Control Panel. If you select the General tab, you can change your Locale, and the default date and time formatting will reflect the preferences in that location. You can select the Time and Date tabs to change the default formatting to something you prefer. Date and time display formats that depend on cultural differences or user preferences are said to be \textit{culture-specific}.

While it's often polite for a program to format a date or time in a manner that is recognizable and readable by the user, sometimes it's undesirable. Sometimes dates and times must be embedded in documents that must be viewed by people in other cultures or merged with similar documents. In this
case, a program should use a consistent date and time format, perhaps in accordance with some international standard. In the jargon of the .NET Framework, such formats are said to be culture-invariant.

For the following examples, I'll be using a day of June 1, 2002, and a local time of 3:05:01 p.m. The single-digit month, day, hour, minute, and second will help clarify whether any zero-suppression is going on in the formatting.

Let's assume that dt is a DateTime object. If you simply put this object in a Console.WriteLine method, as

```csharp
Console.WriteLine(dt);
```

or in a String.Format method, as

```csharp
str = String.Format("{(0)}", dt);
```

the method causes the ToString method of DateTime to be called. The previous statement is equivalent to

```csharp
str = dt.ToString();
```

ToString converts the date and time to a culture-specific character string. For U.S. English settings, the character string returned from ToString looks like this:

6/1/2002 3:05:01 PM

The DateTime structure also defines several additional versions of ToString that have one or two arguments. These versions allow you to format the date and time in a variety of culture-specific and culture-invariant ways:

**DateTime ToString Method**

```csharp
string ToString()
string ToString(string strFormat)
string ToString(IFormatProvider ifp)
string ToString(string strFormat, IFormatProvider ifp)
```

The string argument is typically a single letter that denotes a particular style of formatting. I'll be describing these letters in detail shortly. The string argument can also be a series of letters that describe a custom format.

The IFormatProvider argument refers to an interface. What you need for this argument is an instance of a class that implements IFormatProvider. One such class is DateTimeFormatInfo, which is in the System.Globalization namespace. (You should check the documentation for DateTimeFormatInfo if you need formatting information beyond what I'm presenting here.) The DateTimeFormatInfo class has two static properties, both of which return instances of the class:

**DateTimeFormatInfo Static Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTimeFormatInfo</td>
<td>CurrentInfo</td>
<td>get</td>
</tr>
<tr>
<td>DateTimeFormatInfo</td>
<td>InvariantInfo</td>
<td>get</td>
</tr>
</tbody>
</table>

Thus, to get culture-invariant formatting, you can call

```csharp
strDT = dt.ToString(DateTimeFormatInfo.InvariantInfo);
```

or

```csharp
strDT = dt.ToString(strFormat, DateTimeFormatInfo.InvariantInfo);
```
To get formatting consistent with the user's cultural and personal preferences as specified in Control Panel, use

```
strDT = dt.ToString(DateTimeFormatInfo.CurrentInfo);
```

or

```
strDT = dt.ToString(strFormat, DateTimeFormatInfo.CurrentInfo);
```

You'll also get culture-specific formatting if you use `null` as the second argument to `ToString`:

```
strDT = dt.ToString(strFormat, null);
```

or if you use the version with `strFormat` as the only argument:

```
strDT = dt.ToString(strFormat);
```

To use standard date and time formats, set the first argument of `ToString` to a single character as shown in the first column of the following table. The second column in this table displays the formatting you get when you've used Control Panel to set the locale to the United States, and you've specified `null` or `DateTimeFormatInfo.CurrentInfo` as the second argument to `ToString` or you've used the version of `ToString` that has a single string argument. The column on the right shows the formatted strings when you use a second argument of `DateTimeFormatInfo.InvariantInfo`. These strings are the same regardless of Control Panel settings.

<table>
<thead>
<tr>
<th><code>ToString Date and Time Formats</code></th>
<th><code>Format Date and Time Formats</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td><strong>Format Argument</strong></td>
</tr>
<tr>
<td><strong>Character</strong></td>
<td><strong>CurrentInfo for United States</strong></td>
</tr>
<tr>
<td><code>d</code></td>
<td>6/1/2002</td>
</tr>
<tr>
<td><code>D</code></td>
<td>Saturday, June 01, 2002</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Saturday, June 01, 2002 3:05 PM</td>
</tr>
<tr>
<td><code>F</code></td>
<td>Saturday, June 01, 2002 3:05:01 PM</td>
</tr>
<tr>
<td><code>g</code></td>
<td>6/1/2002 3:05 PM</td>
</tr>
<tr>
<td><code>G</code> or <code>null</code></td>
<td>6/1/2002 3:05:01 PM</td>
</tr>
<tr>
<td><code>m</code> or <code>M</code></td>
<td>June 01</td>
</tr>
<tr>
<td><code>r</code> or <code>R</code></td>
<td>Sat, 01 Jun 2002 15:05:01 GMT</td>
</tr>
<tr>
<td><code>s</code></td>
<td>2002-06-01T15:05:01</td>
</tr>
<tr>
<td><code>t</code></td>
<td>3:05 PM</td>
</tr>
<tr>
<td><code>T</code></td>
<td>3:05:01 PM</td>
</tr>
<tr>
<td><code>u</code></td>
<td>2002-06-01 15:05:01Z</td>
</tr>
<tr>
<td><code>U</code></td>
<td>Saturday, June 01, 2002 7:05:01 PM</td>
</tr>
<tr>
<td><code>y</code> or <code>Y</code></td>
<td>June, 2002</td>
</tr>
</tbody>
</table>

The letters are mnemonics of sorts:

<table>
<thead>
<tr>
<th><code>DateTime Formatting Mnemonics</code></th>
<th><code>Mnemonic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Letter</strong></td>
<td><strong>Mnemonic</strong></td>
</tr>
<tr>
<td><code>d</code></td>
<td>Date</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Full</td>
</tr>
<tr>
<td><code>g</code></td>
<td>General</td>
</tr>
<tr>
<td><code>m</code></td>
<td>month/day</td>
</tr>
</tbody>
</table>
When the uppercase and lowercase letter produce different results (such as \textit{d} and \textit{D}), the uppercase letter produces a longer string. For the \textit{r}, \textit{R}, \textit{s}, or \textit{u} formatting strings, the results are the same regardless of the second argument to \textit{ToString}.

The \textit{ToString} method with a \texttt{null} or an absent string argument returns a string corresponding to the culture-specific \texttt{G} (general) formatting option. Thus, the call
\begin{verbatim}
dt.ToString()
\end{verbatim}
is also equivalent to
\begin{verbatim}
dt.ToString((string)null)
\end{verbatim}
or
\begin{verbatim}
dt.ToString((IFormatProvider)null)
\end{verbatim}
or
\begin{verbatim}
dt.ToString(null, null);
\end{verbatim}
All return culture-specific \texttt{G} formatted strings. The \textit{ToString} methods with a single \texttt{IFormatProvider} argument also return strings equivalent to the \texttt{G} formatting option.

Using \textit{r} or \textit{R} results in the RFC 1123\cite{RFC1123} format. The \textit{s} format is known as ISO 8601\cite{ISO8601} format, and it is intended to be universal and easily sortable. The \textit{T} in the center is known as a \textit{time designator} and separates the date and time. Dates that begin with months or days of the month can't be sorted quite as easily in this format. The \textit{u} formatting is quite similar to \textit{s} except that the time designator is missing and the string ends with a \texttt{Z}. In military and radio circles, UTC is sometimes known as \textit{Zulu time}, \textit{Zulu} being used to represent \texttt{Z}, and \texttt{Z} referring to zero degrees of longitude.

The \textit{U} format option performs a conversion to UTC. The use of this formatting string implies that the \textit{DateTime} value is a local time.

The \textit{DateTime} structure has four other convenient formatting methods, all of which are culture-specific:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Method} & \textbf{Resultant String (U.S. English)} \\
\hline
\texttt{string ToShortDateString()} & 6/1/2001 \\
\hline
\texttt{string ToLongDateString()} & Saturday, June 01, 2002 \\
\hline
\texttt{string ToShortTimeString()} & 3:05 PM \\
\hline
\texttt{string ToLongTimeString()} & 3:05:01 PM \\
\hline
\end{tabular}
\end{table}

These are identical to the culture-specific formatting strings of \textit{d}, \textit{D}, \textit{t}, and \textit{T}, respectively.

Just for kicks, let's go into the Regional Options dialog box in Control Panel and change the locale to Germany. Now let's look at how date and time strings are formatted with default German settings:
**ToString Date and Time Formats (German)**

<table>
<thead>
<tr>
<th>Format</th>
<th>Format Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>CurrentInfo for Germany</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>01.06.2002</td>
</tr>
<tr>
<td>&quot;D&quot;</td>
<td>Samstag, 1. Juni 2002</td>
</tr>
<tr>
<td>&quot;f&quot;</td>
<td>Samstag, 1. Juni 2002 15:05</td>
</tr>
<tr>
<td>&quot;F&quot;</td>
<td>Samstag, 1. Juni 2002 15:05:01</td>
</tr>
<tr>
<td>&quot;g&quot;</td>
<td>01.06.2002 15:05</td>
</tr>
<tr>
<td>&quot;G or null&quot;</td>
<td>01.06.2002 15:05:01</td>
</tr>
<tr>
<td>&quot;m or M&quot;</td>
<td>01 Juni</td>
</tr>
<tr>
<td>&quot;r or R&quot;</td>
<td>Sat, 01 Jun 2002 15:05:01 GMT</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>2002-06-01T15:05:01</td>
</tr>
<tr>
<td>&quot;t&quot;</td>
<td>15:05</td>
</tr>
<tr>
<td>&quot;T&quot;</td>
<td>15:05:01</td>
</tr>
<tr>
<td>&quot;u&quot;</td>
<td>2002-06-01 15:05:01Z</td>
</tr>
<tr>
<td>&quot;U&quot;</td>
<td>Samstag, 1. Juni 2002 19:05:01</td>
</tr>
<tr>
<td>&quot;y or Y&quot;</td>
<td>Juni 2002</td>
</tr>
</tbody>
</table>

The column on the right is the same as the previous table. The culture-specific formatting involves using German names for months and days of the week as well as other formatting specifics.

In some cases, `ToString` does more than just format. If you change your locale to one of the Arab countries, you have a choice (on the Date tab of the Regional Options dialog box) between using the Gregorian calendar or the Islamic calendar. Likewise, if you select Hebrew in the Regional Options, you can choose between the Gregorian calendar or the Hebrew calendar. If you choose an Islamic or Hebrew calendar, the culture-specific format options will perform a conversion to a date in that calendar.

RFC stands for "request for comment" and is the means by which Internet standards are distributed. The time and date specification in RFC 1123 slightly modifies the specification discussed in RFC 822. RFCs are available at many Web sites, including [http://www.ietf.org](http://www.ietf.org).

ISO 8601 ("Date elements and interchange formats—Information interchange—Representation of dates and times") is available from the ISO at [http://www.iso.ch](http://www.iso.ch). ISO 8601 is actually a collection of formats for representing dates and times. The format used by the `ToString` method of `DateTime` is the first of the three extended formats shown in section 5.4.1.a.

**A Simple Culture-Specific Clock**

Here's a program that uses the `F` formatting option to display the current date and time in its client area.

**SimpleClock.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleClock : Form
```
{ public static void Main()
{
    Application.Run(new SimpleClock());
}
public SimpleClock()
{
    Text = "Simple Clock";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    Timer timer = new Timer();
timer.Tick += new EventHandler(TimerOnTick);
timer.Interval = 1000;
timer.Start();
}
private void TimerOnTick(object sender, EventArgs ea)
{
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    StringFormat strfmt = new StringFormat();
    strfmt.Alignment = StringAlignment.Center;
    strfmt.LineAlignment = StringAlignment.Center;

                            Font, new SolidBrush(ForeColor),
                            ClientRectangle, strfmt);
}

The program sets a 1-second timer and simply invalidates the client area in response to the OnTick event. Here's what it looks like with the default U.S. formatting in effect:
But I ask you: Did we spend all that time learning about fonts only to create a clock as pathetic as this one? I don’t think so.

Let’s restrict ourselves to the time and make it as large as possible. That just requires using `OnPaint` processing more creatively.

`DigitalClock.cs`

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class DigitalClock : Form
{
    public static void Main()
    {
        Application.Run(new DigitalClock());
    }
    public DigitalClock()
    {
        Text = "Digital Clock";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
        MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);
    }
}
```

---

Friday, March 30, 2001 12:50:21 PM

---
Timer timer = new Timer();
timer.Tick += new EventHandler(TimerOnTick);
timer.Interval = 1000;
timer.Start();
}
private void TimerOnTick(object obj, EventArgs ea)
{
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    string strTime = DateTime.Now.ToString("T");
    SizeF sizef = grfx.MeasureString(strTime, Font);
    float fScale = Math.Min(ClientSize.Width / sizef.Width,
                             ClientSize.Height / sizef.Height);
    Font font = new Font(Font.FontFamily,
                          fScale * Font.SizeInPoints);

    sizef = grfx.MeasureString(strTime, font);

    grfx.DrawString(strTime, font, new SolidBrush(ForeColor),
                    (ClientSize.Width - sizef.Width) / 2,
                    (ClientSize.Height - sizef.Height) / 2);
}

The OnPaint method stores the formatted time in strTime and then uses a technique I discussed in Chapter 9 to make the text as large (but no larger) than the client area. Here's the display with default U.S. English settings:
Unfortunately, for the sake of getting something large enough to read from across the room, we've lost the date display. Is it possible to display both the date and the time while maintaining the big size? Of course! The trick is to avoid using those format strings that combine the date and time, and to format the date and time separately, combining the two strings with a line feed character.

DigitalClockWithDate.cs

//---------------------------------------------------
// DigitalClockWithDate.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DigitalClockWithDate: DigitalClock
{
    public new static void Main()
    {
        Application.Run(new DigitalClockWithDate());
    }
    public DigitalClockWithDate()
    {
        Text += " with Date";
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        DateTime dt = DateTime.Now;
        string strTime = dt.ToString("d") + "\n" + dt.ToString("T");
        SizeF sizef = grfx.MeasureString(strTime, Font);
        float fScale = Math.Min(ClientSize.Width / sizef.Width, ClientSize.Height / sizef.Height);
        Font font = new Font(Font.FontFamily, fScale * Font.SizeInPoints);

        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString(strTime, font, new SolidBrush(ForeColor), ClientRectangle, strfmt);
    }
}

The call to MeasureString returns the height of the two lines of text and the width of the wider line. To center both lines horizontally in the client area, the DrawString call needs a StringFormat object. Here's the display, again with default U.S. English settings:
The fact that the `OnPaint` method gets called every second might get you to wondering about the efficiency of the drawing code. Would it be better, for example, to create the required font during the `OnResize` method? Yes, it would, but it would take a bit of work to get it to work just right. The font size is dependent on both the size of the client area and the width and height of the text string. In most cases, of course, the text string width doesn’t change from second to second. But it does change occasionally. The width of the text string containing the time depends on whether the hour is one digit or two digits wide. And if the time is displayed in a 24-hour format, the date string is wider than the time string, and that width depends on the month and the day of the month.

If the program had a separate method to create an appropriate font, that method would have to retain in fields both the font and the width of the text used to calculate the font. The `OnResize` method would call the font-calculation method, obviously. And the `OnPaint` method would need to call it as well if the text width didn’t match the width used to calculate the font.

**The Retro Look**

You could use any TrueType font you have installed on your system in either of the digital clock programs. Just put a statement in the program’s constructor to change the form’s `Font` property:

```csharp
Font = new Font("Comic Sans MS", 12);
```

Then just let the `OnPaint` method scale the font.

To give the clock a neat retro look, you might want to choose a font that looks like a seven-segment LCD display. Or you can use this `SevenSegmentDisplay` class instead of a font.

```csharp
//--------------------------------------------------
// SevenSegmentDisplay.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
namespace Petzold.ProgrammingWindowsWithCSharp
{
    class SevenSegmentDisplay
    {
        Graphics grfx;
    }
```
// Indicates what segments are illuminated for all 10 digits

static byte[,] bySegment = {{1, 1, 1, 0, 1, 1, 1},       // 0
                           {0, 0, 1, 0, 0, 1, 0},       // 1
                           {1, 0, 1, 1, 1, 0, 1},       // 2
                           {1, 0, 1, 1, 0, 1, 1},       // 3
                           {0, 1, 1, 1, 0, 1, 0},       // 4
                           {1, 1, 0, 1, 0, 1, 1},       // 5
                           {1, 1, 0, 1, 1, 1, 1},       // 6
                           {1, 0, 1, 0, 0, 1, 0},       // 7
                           {1, 1, 1, 1, 1, 1, 1},       // 8
                           {1, 1, 1, 1, 0, 1, 1}};      // 9

// Points that define each of the seven segments

readonly Point[][] apt = new Point[7][];

public SevenSegmentDisplay(Graphics grfx)
{
    this.grfx = grfx;
    // Initialize jagged Point array.

    apt[0] = new Point[] {new Point( 3,  2), new Point(39,  2),
                           new Point(31, 10), new Point(11, 10)};

    apt[1] = new Point[] {new Point( 2,  3), new Point(10, 11),
                           new Point(10, 31), new Point( 2, 35)};

    apt[2] = new Point[] {new Point(40,  3), new Point(40, 35),
                           new Point(32, 31), new Point(32, 11)};

    apt[3] = new Point[] {new Point( 3, 36), new Point(11, 32),
                           new Point(31, 32), new Point(39, 36),
                           new Point(31, 40), new Point(11, 40)};

    apt[4] = new Point[] {new Point( 2, 37), new Point(10, 41),
                           new Point(10, 61), new Point( 2, 69)};

    apt[5] = new Point[] {new Point(40, 37), new Point(40, 69),
                           new Point(32, 61), new Point(32, 41)};

    apt[6] = new Point[] {new Point(11, 62), new Point(31, 62),
                           new Point(39, 70), new Point( 3, 70)};
}

public SizeF MeasureString(string str, Font font)
SizeF sizef = new SizeF(0, grfx.DpiX * font.SizeInPoints / 72);

for (int i = 0; i < str.Length; i++)
{
    if (Char.IsDigit(str[i]))
        sizef.Width += 42 * grfx.DpiX * font.SizeInPoints / 72 / 72;
    else if (str[i] == ':')
        sizef.Width += 12 * grfx.DpiX * font.SizeInPoints / 72 / 72;
}
return sizef;

public void DrawString(string str, Font font, Brush brush, float x, float y)
{
    for (int i = 0; i < str.Length; i++)
    {
        if (Char.IsDigit(str[i]))
            x = Number(str[i] - '0', font, brush, x, y);
        else if (str[i] == ':')
            x = Colon(font, brush, x, y);
    }
}
float Number(int num, Font font, Brush brush, float x, float y)
{
    for (int i = 0; i < apt.Length; i++)
        if (bySegment[num, i] == 1)
            Fill(apt[i], font, brush, x, y);
    return x + 42 * grfx.DpiX * font.SizeInPoints / 72 / 72;
}
float Colon(Font font, Brush brush, float x, float y)
{
    Point[][] apt = new Point[2][];
    apt[0] = new Point[] {new Point( 2, 21), new Point( 6, 17),
                          new Point(10, 21), new Point( 6, 25)};
    apt[1] = new Point[] {new Point( 2, 51), new Point( 6, 47),
                          new Point(10, 51), new Point( 6, 55)};
The **SevenSegmentDisplay** class has one public constructor, which takes an argument of type *Graphics*, and two public methods, which are called *MeasureString* and *DrawString* and which have the same arguments as the two most popular versions of those methods in the *Graphics* class. The idea here is that you create a *SevenSegmentDisplay* object with a *Graphics* object argument and then use these two methods instead of the methods in the *Graphics* class.

The *DrawString* method implemented in *SevenSegmentDisplay* can deal with only 11 character codes: those for the 10 digits and the colon. It calls the private *Number* and *Colon* methods for these two cases. The *Number* method uses a static array named *bySegment* that indicates which of the seven segments are illuminated for each of the 10 digits. (This array should probably have been defined with the *bool* data type rather than *byte*, but I thought that the list of *true* and *false* initializers would have been more difficult to read, and I couldn't imagine that the machine code would have been more efficient.) A *readonly* jagged *Point* array named *apt* has the points that define each of the seven segments. These points are based on a character width of 42 and a height of 72. The private *Fill* method scales these coordinates based on the font size and uses *FillPolygon* to color the interiors red.

The clock program that uses this class is virtually identical to DigitalClock except that it begins *OnPaint* processing by creating a *SevenSegmentDisplay* object and uses that rather than the *Graphics* object for calls to *MeasureString* and *DrawString*.

**SevenSegmentClock.cs**

```csharp
using System;
using System.Drawing;
using System.Globalization;
using System.Windows.Forms;
using Petzold.ProgrammingWindowsWithCSharp;
```
class SevenSegmentClock: Form
{
    DateTime dt;

    public static void Main()
    {
        Application.Run(new SevenSegmentClock());
    }
    public SevenSegmentClock()
    {
        Text = "Seven-Segment Clock";
        BackColor = Color.White;
        ResizeRedraw = true;
        MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);
        dt = DateTime.Now;

        Timer timer = new Timer();
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Interval = 100;
        timer.Enabled = true;
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        DateTime dtNow = DateTime.Now;
        dtNow = new DateTime(dtNow.Year, dtNow.Month, dtNow.Day,
                              dtNow.Hour, dtNow.Minute, dtNow.Second);
        if (dtNow != dt)
        {
            dt = dtNow;
            Invalidate();
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        SevenSegmentDisplay ssd = new SevenSegmentDisplay(pea.Graphics);
        string strTime = dt.ToString("T",
                                      DateTimeFormatInfo.InvariantInfo);
        SizeF sizef = ssd.MeasureString(strTime, Font);
        float fScale = Math.Min(ClientSize.Width / sizef.Width,
                                          ClientSize.Height / sizef.Height);
        Font font = new Font(Font.FontFamily,
                                fScale * Font.SizeInPoints);
sizef = ssd.MeasureString(strTime, font);

ssd.DrawString(strTime, font, Brushes.Red,
    (ClientSize.Width - sizef.Width) / 2,
    (ClientSize.Height - sizef.Height) / 2);
}
}

However, notice that I've used a culture-invariant ToString method of DateTime. This is an excellent example of a program that works best with a culture-invariant string because it needs to know exactly what characters it's getting and doesn't want to encounter a.m. or p.m. indicators:

An Analog Clock

Digital clocks were popular when they were new, but the pendulum has swung back (so to speak) to analog clocks. An analog clock needn't concern itself with different date and time formats, but the complexity of the graphics more than outweighs that convenience. Users have come to expect analog clocks to dynamically change size with the size of the window.

For greatest versatility, I decided to write the clock display logic as a child window control like the CheckerChild class in the CheckerWithChildren program in Chapter 8. That would make it easier to embed a clock display in another application or to write an application that displayed multiple clocks. Here's the code for the ClockControl class.

ClockControl.cs
//------------------------------------------------------------------------------
// ClockControl.cs © 2001 by Charles Petzold
//------------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
namespace Petzold.ProgrammingWindowsWithCSharp
{
    class ClockControl: UserControl

DateTime dt;

public ClockControl()
{
    ResizeRedraw = true;
    Enabled = false;
}

public DateTime Time
{
    get
    {
        return dt;
    }
    set
    {
        Graphics grfx = CreateGraphics();
        InitializeCoordinates(grfx);

        Pen pen = new Pen(BackColor);

        if (dt.Hour != value.Hour)
        {
            DrawHourHand(grfx, pen);
        }
        if (dt.Minute != value.Minute)
        {
            DrawHourHand(grfx, pen);
            DrawMinuteHand(grfx, pen);
        }
        if (dt.Second != value.Second)
        {
            DrawMinuteHand(grfx, pen);
            DrawSecondHand(grfx, pen);
        }
        if (dt.Millisecond != value.Millisecond)
        {
            DrawSecondHand(grfx, pen);
        }
        dt = value;
        pen = new Pen(ForeColor);

        DrawHourHand(grfx, pen);
        DrawMinuteHand(grfx, pen);
        DrawSecondHand(grfx, pen);
grfx.Dispose();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Pen pen = new Pen(ForeColor);
    Brush brush = new SolidBrush(ForeColor);

    InitializeCoordinates(grfx);
    DrawDots(grfx, brush);
    DrawHourHand(grfx, pen);
    DrawMinuteHand(grfx, pen);
    DrawSecondHand(grfx, pen);
}

void InitializeCoordinates(Graphics grfx)
{
    if (Width == 0 || Height == 0)
        return;

    grfx.TranslateTransform(Width / 2, Height / 2);

    float fInches = Math.Min(Width / grfx.DpiX, Height / grfx.DpiY);
    grfx.ScaleTransform(fInches * grfx.DpiX / 2000,
                     fInches * grfx.DpiY / 2000);
}

void DrawDots(Graphics grfx, Brush brush)
{
    for (int i = 0; i < 60; i++)
    {
        int iSize = i % 5 == 0 ? 100 : 30;
        grfx.FillEllipse(brush, 0 - iSize / 2, -900 - iSize / 2,
                         iSize, iSize);
        grfx.RotateTransform(6);
    }
}

protected virtual void DrawHourHand(Graphics grfx, Pen pen)
{
    GraphicsState gs = grfx.Save();
    grfx.RotateTransform(360f * Time.Hour / 12 +
                          30f * Time.Minute / 60);
grfx.DrawPolygon(pen, new Point[]
    {
        new Point(0, 150), new Point(100, 0),
        new Point(0, -600), new Point(-100, 0)
    });

grfx.Restore(gs);
}

protected virtual void DrawMinuteHand(Graphics grfx, Pen pen)
{
    GraphicsState gs = grfx.Save();
    grfx.RotateTransform(360f * Time.Minute / 60 +
        6f * Time.Second / 60);
    grfx.DrawPolygon(pen, new Point[]
        {
            new Point(0, 200), new Point(50, 0),
            new Point(0, -800), new Point(-50, 0)
        });
    grfx.Restore(gs);
}

protected virtual void DrawSecondHand(Graphics grfx, Pen pen)
{
    GraphicsState gs = grfx.Save();
    grfx.RotateTransform(360f * Time.Second / 60 +
        6f * Time.Millisecond / 1000);
    grfx.DrawLine(pen, 0, 0, 0, -800);
    grfx.Restore(gs);
}

ClockControl inherits from UserControl and overrides the OnPaint method. The ClockControl constructor sets the ResizeRedraw control style to true and also sets its Enabled property to false. ClockControl doesn't have any need for keyboard or mouse input, so any such input will pass through to the control's parent.

Notice the private DateTime field I've named dt and the read/write public property named Time that gives other objects access to this field. The control doesn't implement its own timer and doesn't set this property itself; the control simply displays the time indicated by the current value of its Time property. Keeping that Time property up to date is the responsibility of whatever class creates an instance of ClockControl.

The code implementing the set accessor of the Time property seems inordinately lengthy. The temptation, of course, is to simplify the set accessor like so:

dt = value;
Invalidate();

That Invalidate call would cause the control to get an OnPaint call, at which time it would redraw the clock. Visually, however, this simplification is a disaster. The Invalidate call causes the background of the control to be erased and the entire clock must be redrawn. That causes an annoying flickering
of the image. Instead, I've taken a more attractive approach. Let me come back to the set accessor of `Time` after I've discussed the `OnPaint` processing.

`OnPaint` creates a pen and a brush based on the control's foreground color and then calls five other methods. First, `InitializeCoordinates` sets up a coordinate system with an origin at the center of the control and isotropic coordinates that extend to 1000 units in all four directions.

Second, `DrawDots` draws the dots that indicate the minutes and hours. This method uses the `Graphics` class methods `FillEllipse` to draw a dot at 12:00 and `RotateTransform` to rotate 6º for the next dot. The `DrawHourHand`, `DrawMinuteHand`, and `DrawSecondHand` methods also use `RotateTransform`. I've made these three methods virtual functions so that they can be overridden at some point (by a program in Chapter 13, to be precise).

The actual drawing code (`DrawPolygon` for the hour and minute hands and `DrawLine` for the second hand) assumes that the hands are pointing straight up. The call to `RotateTransform` before the drawing code rotates the hand to its proper position. Each of the hand-drawing routines makes a call to the `Save` method of the `Graphics` class to save the current graphics state before calling `RotateTransform`, and `Restore` after it's finished.

Notice that the position of the hour hand is based on both the `Hour` and `Minute` properties of the `DateTime` structure, the position of the minute hand is based on both `Minute` and `Second`, and the position of the second hand is based on the `Second` and `Millisecond` properties. Thus, the hands sweep continuously rather than jump in discrete steps.

Now we're ready to look at the set accessor code of the `DateTime` property. After calling `CreateGraphics` to obtain a `Graphics` object for the control, a call to `InitializeCoordinates` sets up the proper coordinate system. Then the code creates a pen based on the control's background color. What it needs to do is effectively erase any hand that is changing position. The problem, however, is that drawing a particular hand in a background color might also affect one of the other two hands. For that reason, all three hands must be redrawn using the foreground color. Even though there's still a lot of drawing whenever the time changes, this process reduces flickering considerably.

Now that we have a control, implementing a form that uses this control is fairly easy.

```csharp
// AnalogClock.cs
//-------------------------------
// AnalogClock.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
using Petzold.ProgrammingWindowsWithCSharp;

class AnalogClock: Form
{
    ClockControl clkctl;

    public static void Main()
    {
        Application.Run(new AnalogClock());
    }
    public AnalogClock()
    {
        Text = "Analog Clock";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
}
```
In the constructor, the program creates an object of type ClockControl, sets the Parent property of the control to the form, and also initializes the form's Time property to the current date and time.

Next, the form sets a control property I haven't mentioned yet, which is named Dock. This property is implemented in Control, and I'll discuss it in much more detail in Chapter 12. For now, be aware that setting the Dock style of a control to DockStyle.Fill causes the control to fill up the entire display surface of its parent. The clock control will be automatically sized and resized to fit in the form's client area.

The last two properties of the clock control that I set are BackColor to black and ForeColor to white just to make a point that the control isn't controlling its color. The parent has control over that. Of course, it doesn't hurt that a white-on-black clock looks pretty cool:

![Analog Clock](image)

The constructor processing concludes with setting the timer to an interval of 100 milliseconds (1/10 second). Clocks normally need just a 1-second update, but with this one, the second hand wouldn't give the appearance of sweeping continuously if it weren't updated more frequently. The
TimerOnTick event handler simply sets the Time property of the clock control to the current date and time.

A program that didn't want a continuously sweeping second hand would set the timer to 1000 milliseconds and set the Time property of the clock control with a DateTime object that had a Millisecond property of 0. Because the Millisecond property is read-only, this job requires re-creating the DateTime object. The TimerOnTick code would look like this:

```csharp
DateTime dt = DateTime.Now;
dt = new DateTime(dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second);
clkctl.Time = dt;
```

There are other ways to demonstrate that the time displayed by the clock control is entirely governed by the parent. Try replacing the TimerOnTick code with this:

```csharp
clkctl.Time += new TimeSpan(10000000);
```

The clock begins at the correct time but then moves at 10 times the normal speed. Or try this one:

```csharp
clkctl.Time -= new TimeSpan(1000000);
```

The clock moves at normal speed but backward.

**A Little Puzzle Called Jeu de Taquin**

It's now time to program a game. Well, more like a puzzle. This particular puzzle was invented in the 1870s, probably by the famous American puzzle-maker Sam Loyd (1841–1911). For a while, this puzzle was all the rage, particularly in Europe, and was known under various names, including the 15-puzzle, the 14-15 puzzle, and (in France) Jeu de Taquin, the "teasing game."

In its classic form, the puzzle consists of 15 square blocks numbered 1 through 15. The squares are arranged in a 4-by-4 grid, leaving one blank space. You can move the squares around the grid by shifting a square horizontally or vertically into the blank space, which in turn opens a different blank space.

As Sam Loyd presented it, the numbered squares were arranged in consecutive order except with the 14 and 15 reversed. He offered $1000 to anyone who could find a way to shift the squares around to correct the order of the 14 and 15. No one collected the reward because, from that starting point, the puzzle is insolvable.⁴

In computer form, this puzzle was one of the first game programs created for the Apple Macintosh, where it was called PUZZLE. It also appeared in early versions of the Microsoft Windows Software Development Kit (SDK) under the name MUZZLE, where it was the only sample program in the SDK coded in Microsoft Pascal rather than C. Both these programs initially displayed the 15 squares in consecutive order and presented a menu option to scramble the squares. You then attempted to restore the order of the squares or put them into different orders, such as going down the columns rather than across the rows. Because we haven't covered menus yet, my version of the program scrambles the squares when it first starts up. (That's where the timer comes into play.)

The tiles are child windows, but they set their Enabled property to false to let the parent process all keyboard and mouse input. Normally, controls indicate that they're disabled by graying their text, but they don't have to use this approach. In this case, they don't. The OnPaint method uses normal control colors to draw a 3D-like edge.

**JeuDeTaquinTile.cs**

```csharp
// JeuDeTaquinTile.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;
```
class JeuDeTaquinTile: UserControl
{
    int iNum;

    public JeuDeTaquinTile(int iNum)
    {
        this.iNum = iNum;
        Enabled = false;
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        grfx.Clear(SystemColors.Control);

        int cx = Size.Width;
        int cy = Size.Height;
        int wx = SystemInformation.FrameBorderSize.Width;
        int wy = SystemInformation.FrameBorderSize.Height;

        grfx.FillPolygon(SystemBrushes.ControlLightLight,
            new Point[] { new Point(0, cy), new Point(0, 0),
                            new Point(cx, 0), new Point(cx - wx, wy),
                            new Point(wx, wy), new Point(wx, cy - wy)});

        grfx.FillPolygon(SystemBrushes.ControlDark,
            new Point[] { new Point(cx, 0), new Point(cx, cy),
                            new Point(0, cy), new Point(wx, cy - wy),
                            new Point(cx - wx, cy - wy),
                            new Point(cx - wx, wy)});

        Font font = new Font("Arial", 24);
        StringFormat strfmt = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString(iNum.ToString(), font,
            SystemBrushes.ControlText, ClientRectangle, strfmt);
    }
}

The program that creates these tiles and moves them around the grid is a bit more complicated. It
creates the tile controls (and sizes the client area based on those controls) in an override of the
OnLoad method implemented in the Form class. The OnLoad method is called soon before the form
is first displayed; my experience indicates that obtaining Graphics objects and setting the size of a
client area usually works better when done during OnLoad rather than during the constructor.
OnLoad processing concludes with a call to the protected method Randomize, which uses a timer to scramble the tiles.

**JeuDeTaquin.cs**

```csharp
//JeuDeTaquin.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class JeuDeTaquin: Form
{
    const int nRows = 4;
    const int nCols = 4;
    Size sizeTile;
    JeuDeTaquinTile[,] atile = new JeuDeTaquinTile[nRows, nCols];
    Random rand;
    Point ptBlank;
    int iTimerCountdown;

    public static void Main()
    {
        Application.Run(new JeuDeTaquin());
    }

    public JeuDeTaquin()
    {
        Text = "Jeu de Taquin";
        BorderStyle = FormBorderStyle.Fixed3D;
    }

    protected override void OnLoad(EventArgs ea)
    {
        // Calculate the size of the tiles and the form.

        Graphics grfx = CreateGraphics();

        sizeTile = new Size((int)(2 * grfx.DpiX / 3),
                            (int)(2 * grfx.DpiY / 3));
        ClientSize = new Size(nCols * sizeTile.Width,
                              nRows * sizeTile.Height);
        grfx.Dispose();

        // Create the tiles.

        for (int iRow = 0; iRow < nRows; iRow++)
            for (int iCol = 0; iCol < nCols; iCol++)
                {
```
```csharp
int iNum = iRow * nCols + iCol + 1;

if (iNum == nRows * nCols)
    continue;

JeuDeTaquinTile tile = new JeuDeTaquinTile(iNum);
tile.Parent = this;
tile.Location = new Point(iCol * sizeTile.Width, iRow * sizeTile.Height);
tile.Size = sizeTile;

atile[iRow, iCol] = tile;
}
ptBlank = new Point(nCols - 1, nRows - 1);

Randomize();
}
protected void Randomize()
{
    rand = new Random();
iTimerCountdown = 64 * nRows * nCols;

    Timer timer   = new Timer();
timer.Tick     += new EventHandler(TimerOnTick);
timer.Interval = 1;
timer.Enabled  = true;
}
void TimerOnTick(object obj, EventArgs ea)
{
    int x = ptBlank.X;
    int y = ptBlank.Y;

    switch(rand.Next(4))
    {
    case 0:  x++;  break;
    case 1:  x--;  break;
    case 2:  y++;  break;
    case 3:  y--;  break;
    }
    if (x >= 0 && x < nCols && y >= 0 && y < nRows)
        MoveTile(x, y);

    if (--iTimerCountdown == 0)
    {
        ((Timer)obj).Stop();
    }
```
((Timer)obj).Tick -= new EventHandler(TimerOnTick);
}
}
protected override void OnKeyDown(KeyEventArgs kea)
{
    if (kea.KeyCode == Keys.Left && ptBlank.X < nCols - 1)
        MoveTile(ptBlank.X + 1, ptBlank.Y);

    else if (kea.KeyCode == Keys.Right && ptBlank.X > 0)
        MoveTile(ptBlank.X - 1, ptBlank.Y);

        MoveTile(ptBlank.X, ptBlank.Y + 1);

    else if (kea.KeyCode == Keys.Down && ptBlank.Y > 0)
        MoveTile(ptBlank.X, ptBlank.Y - 1);

    kea.Handled = true;
}
protected override void OnMouseDown(MouseEventArgs mea)
{
    int x = mea.X / sizeTile.Width;
    int y = mea.Y / sizeTile.Height;

    if (x == ptBlank.X)
    {
        if (y < ptBlank.Y)
            for (int y2 = ptBlank.Y - 1; y2 >= y; y2--)
                MoveTile(x, y2);

        else if (y > ptBlank.Y)
            for (int y2 = ptBlank.Y + 1; y2 <= y; y2++)
                MoveTile(x, y2);
    }

    else if (y == ptBlank.Y)
    {
        if (x < ptBlank.X)
            for (int x2 = ptBlank.X - 1; x2 >= x; x2--)
                MoveTile(x2, y);

        else if (x > ptBlank.X)
            for (int x2 = ptBlank.X + 1; x2 <= x; x2++)
                MoveTile(x2, y);
    }
}
void MoveTile(int x, int y) {
    atile[y, x].Location = new Point(ptBlank.X * sizeTile.Width,
                                        ptBlank.Y * sizeTile.Height);

    atile[ptBlank.Y, ptBlank.X] = atile[y, x];
    atile[y, x] = null;
    ptBlank = new Point(x, y);
}

Everything else in the program just involves processing keyboard and mouse input leading up to a call to the MoveTile method at the bottom of the listing.

The two-dimensional atile array stores the tile objects. For example, the tile object stored at atile[3, 1] is the tile currently in the fourth row and second column of the grid. One element of the atile array is always null. That null element corresponds to the coordinate currently not occupied by any tile. The ptBlank field also stores that coordinate. The blank—as I'll call it—governs the user interface; likewise, ptBlank plays a major role in the user interface code. Any tile that the program moves must be adjacent to the blank, and it must move into the blank.

When you use the mouse, you don't have to click a tile adjacent to the blank, however. If you click a tile in the same row or column as the blank, the program moves multiple tiles with one shot, which means it makes multiple calls to MoveTile. The MoveTile method both physically moves the tile (by setting the Location property of the tile being moved to the location of the blank) and adjusts the atile and ptBlank fields accordingly.

The keyboard interface involves the arrow keys. If you think about it, pressing any of the four arrow keys has an unambiguous meaning. For example, pressing the down key always moves the tile immediately above the blank (if any) into the location of the blank.

Here's a sample view of the program as I'm about halfway through solving it:

This is now the third program in this book in which creating a custom control has been found useful. As you undoubtedly know, Windows and the Windows Forms .NET Framework implement a
multitude of ready-made controls in the form of buttons, labels, text-entry fields, list boxes, scroll bars, and much more. We'll begin exploring that world in Chapter 12.

Chapter 11: Images and Bitmaps

Overview

The world of computer graphics is generally divided into two distinct areas: vector graphics and raster graphics. In mathematics, a vector is a combination of a magnitude and a direction, and it can also refer to a line in a coordinate space. Vector graphics is the application of analytical geometry to draw lines, curves, and filled areas. With the use of outline fonts, text can also be considered part of vector graphics.

The term raster comes from video display technology and refers to the use of multiple scan lines to form a composite image. In raster graphics, images are described by rectangular arrays of pixels known as bitmaps.

Both vector and raster graphics have their origins in graphical display devices. Most display devices in use today are raster devices. In laser or ink jet printers, the image on the page is a rectangular array of pixels that are colored with dots. Cathode ray tubes (CRTs) display images as a collection of horizontal scan lines, each of which is made up of a series of pixels. The bits that define the CRT image are stored in a block of memory on the video display board.

Although raster output devices certainly seem normal today, in the early days of computer graphics (the 1950s), memory was too expensive for raster displays. Video displays attached to a computer worked much like an oscilloscope: the cathode ray didn't draw horizontal scan lines but was instead deflected directly to draw lines and curves under computer control. Vector printing devices were also more common in days gone by and still exist today in the form of plotters.

Vectors and rasters both have their place in the world of graphics. An architectural drawing is obviously a job for vector graphics, whereas a realistic-looking image of what the completed building will look like is a job for raster graphics. Vector and raster graphics are generally the provinces of different types of applications: draw programs do vector graphics, and paint programs do raster graphics. Photo programs are variations of paint programs that work with real-world raster images captured from digital cameras or scanned photographs.

As we've seen, vector images can be subjected to transforms that change their size and orientation. This transformation happens without any loss of resolution. A 10-point font scaled in size by a factor of 10 doesn't accumulate any jaggies at that new size because the vector outlines are being scaled. Raster images, however, usually have device dependencies that can't easily be ignored. Bitmaps have specific pixel sizes. Attempting to display a bitmap in a larger size can result in jaggies; in a smaller size, information can be lost. (GDI+ attempts to minimize these problems by using smoothing algorithms.) Bitmap images also contain specific color information that can't always be rendered on a specific output device.

Converting a vector image to a raster image is very easy. All that's necessary is to draw the various lines, curves, filled areas, and text on the surface of a bitmap. (We'll do this later in the chapter.) Converting a raster image to a vector image can be quite difficult, however, and is feasible only with simple images.

Many older or traditional books on computer graphics focus almost entirely on vector graphics. Today's dominance of raster graphics is a more recent phenomenon brought about by low-cost memory, scanners, and digital cameras. Also helping this trend are bitmap compression technologies, such as JPEG, that help cut down on the memory bulk normally associated with bitmaps.

Almost all the graphics found on the World Wide Web are stored as bitmaps; to many Web users, computer graphics is synonymous with JPEG and GIF files. This is not necessarily a good thing. Many Web graphics would be more efficiently stored and transmitted as vector images, particularly considering that such images often originate as lines, curves, and filled areas in paint programs. However, recent attempts at promoting vector graphics standards for the Web have not caught on much. \[\text{You can learn about the proposed Scalable Vector Graphics (SVG) standard on the Web site of the World Wide Web consortium (www.w3.org/Graphics/SVG), and you can learn about the Vector Markup Language (VML), which is supported by recent versions of Internet Explorer, on the}\]
Microsoft Web site at https://msdn.microsoft.com/workshop/author/vml. Both are XML-based. Finding Web sites that display images using SVG or VML is a much bigger challenge.

**Bitmap Support Overview**

The `System.Drawing` namespace has two classes, named `Image` and `Bitmap`, that provide much of the raster graphics support in .NET. The `Bitmap` class as well as the `Metafile` class (to be covered in Chapter 23) are derived from `Image`, as shown in the following class hierarchy:

```
Object
   MarshalByRefObject
      Image (abstract)
         Bitmap
         Metafile
```

`Image` is an abstract class that can't be instantiated using a constructor. However, `Image` has two static methods (four if you count overloads) that return objects of type `Image`. These methods are extremely powerful, for they can load a bitmap or a metafile from a file or a stream. You probably expect these methods to work with BMP files because that's the native Windows bitmap format. What you'll undoubtedly be pleased to learn is that these methods can also load files in several other popular bitmap formats, including GIF, JPEG, PNG, and TIFF. `Image` also has another static method, which lets you create a `Bitmap` object from a Win32 bitmap handle.

Once you have an `Image` object, you can do a couple things with it. You can display it on the screen or a printer by using one of the `DrawImage` methods in the `Graphics` class. Or you can use the static `FromImage` method of `Graphics` to return a `Graphics` object that applies to the image. This facility lets a program draw on a bitmap. Additional methods of the `Image` class allow you to save an image object in one of the supported formats. Thus, `Image` has a built-in format-conversion facility.

If you just need to load and display bitmap images, the `Image` class is probably all you need. The `Bitmap` class extends `Image` by providing a number of constructors that let you create a new bitmap of a particular size and color format. The `Bitmap` class also allows you to directly read and write individual pixels and to access the bitmap data as a block of memory.

The `Bitmap` class also includes a constructor that lets you load a bitmap that's been embedded in the `.exe` file as a resource. You can also use this technique for loading icons and custom cursors. I'll discuss binary resources later in this chapter.

Sometimes programmers will wonder whether they should use the `Image` or `Bitmap` class for a particular task. If everything you need to do can be done with `Image`, then use `Image`. The bonus is that your code (with some exceptions) will also work with metafiles.

**Bitmap File Formats**

A bitmap is a rectangular array of bits that correspond to the pixels of a graphics output device. A bitmap has a particular height and width measured in pixels. A bitmap also has a particular color depth, which is the number of bits per pixel (commonly abbreviated `bpp`). Each pixel in the bitmap has the same number of bits, which determines how many unique colors are in the image:

\[
\text{Number of colors} = 2^{\text{Number of bits per pixel}}
\]

The number of bits per pixel can generally range from 1 to 32 (and even beyond), although some formats are more common than others.
In graphical environments such as Windows, colors are usually represented as RGB (red-green-blue) values, where each primary is 1 byte and a full RGB color value is 3 bytes, or 24 bits. Such a color resolution seems to be fairly close to the ability of the human eye to differentiate between colors. It also approximates the ability of today's monitors to render distinct colors. An additional byte can represent levels of transparency, ranging from complete opacity to complete transparency. But 32 bits per pixel is not the ultimate limit. Some applications—such as medical imaging—require more bits per pixel for increased resolution.

A bitmap with 1 bit per pixel stores a bilevel, or monochrome, image. Only two colors are possible; these are often black and white, but not always. Generally, such a bitmap contains a small color table (or color palette) that indicates the two colors associated with the two bit values.

In the early days of Windows, 4-bit-per-pixel images were popular, and these can still be found. For example, icons are often 16-color images. The 16 colors are generally combinations of the red, green, and blue primaries in regular and dark versions. Such a bitmap contains a color table indicating the exact colors corresponding to the 16 different possible combinations of pixels.

A very common bitmap format has 8 bits per pixel. Often, the image is gray scale and the 8 bits correspond to 256 (or fewer) gray shades from black to white. However, color images can also be stored in 8-bit-per-pixel bitmaps, in which case the 256 (or fewer) colors are usually specifically chosen for the particular image. This color choice is sometimes known as the optimized palette for the image.

A bitmap with 16 bits per pixel generally uses 5 bits each for the red, green, and blue levels, with 1 bit unused. Thus, each primary can have 32 different values, for a total of 32,768 unique colors. Sometimes green gets an extra bit because it's the color human eyes are most sensitive to. Such a bitmap is sometimes referred to as a 5-6-5, referring to the number of bits used for each primary. The use of 15 or 16 bits per pixel is sometimes referred to as high color and is insufficient to represent color gradations in some real-world images.

A full-color, or true-color, bitmap has 24 bits per pixel. Each pixel is a 24-bit RGB color value. The use of exactly 3 bytes per pixel in a bitmap can result in a performance problem: generally, 32-bit processors are most efficient if they access 32-bit values on 32-bit memory boundaries.

A 32-bit-per-pixel bitmap can actually be a 24 bit-per-pixel image with 1 byte per pixel unused for performance purposes. Or the additional byte could provide transparency information, known as an alpha channel. For each pixel, the alpha value indicates a level of transparency for the pixel.

Bitmaps can be very large. For example, if you run your video adapter in 1600 × 1200 pixel mode, a 24-bit-per-pixel bitmap that occupies the entire screen is over 5 megabytes in size. For this reason, a great deal of research has gone into the development of image compression techniques.

One simple method that occurs to just about everyone who thinks about image compression is called run-length encoding (RLE). If there are 12 consecutive blue pixels, for example, it makes sense to store the number of repeated pixels rather than all 12 pixels. RLE generally works well for images that have a limited number of colors, such as cartoonlike images.

To go beyond RLE, it's necessary for a compression program to analyze the data for recurring patterns. A major advance in data compression occurred in the late 1970s when Jacob Zif and Abraham Lempel published compression techniques now known as LZ77 and LZ78. These algorithms find patterns in the data on the fly and efficiently indicate when the patterns are reused. A 1984 article by Terry Welch of the Sperry Research Center (now part of Unisys) built on LZ78 to form a technique now called LZW. LZW is the basis of several popular compression formats. In recent years, however, Unisys has restricted the unlicensed use of LZW. Given that LZW was already used in several entrenched standards (including GIF), much of the programming community has responded with disdain at the claims of Unisys, and LZW has been deliberately and conspicuously avoided in new compression formats.

RLE and all the LZ techniques are known as lossless compression techniques because the original data can be entirely recovered from the compressed data. (It's fairly simple to prove that a particular lossless compression algorithm can't work for every possible file. For some files, application of the compression algorithm increases the file size!) Lossless compression is essential if you're dealing with spreadsheets or word processing documents. Lossless compression is much less of a concern for real-world images such as digitized photographs.
For that reason, *lossy* compression techniques have become popular in recent years when photographic images must be compressed. Lossy compression works best when it eliminates data that is imperceptible (or less perceptible) to human vision. Extreme levels of lossy compression can result in noticeable degradation of the image, however.

The bitmap file formats supported by the `Image` class are indicated by static properties in `ImageFormat`, a class defined in the `System.Drawing.Imaging` namespace:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Bmp</code></td>
<td><code>get</code></td>
<td>Windows device-independent bitmap (DIB)</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>MemoryBmp</code></td>
<td><code>get</code></td>
<td>Memory-based DIB (no file header)</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Icon</code></td>
<td><code>get</code></td>
<td>Windows icon format</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Gif</code></td>
<td><code>get</code></td>
<td>CompuServe Graphics Interchange Format</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Jpeg</code></td>
<td><code>get</code></td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Png</code></td>
<td><code>get</code></td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Tiff</code></td>
<td><code>get</code></td>
<td>Tag Image File Format</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Exif</code></td>
<td><code>get</code></td>
<td>Exchangeable image format</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Wmf</code></td>
<td><code>get</code></td>
<td>Windows metafile (original)</td>
</tr>
<tr>
<td><code>ImageFormat</code></td>
<td><code>Emf</code></td>
<td><code>get</code></td>
<td>Windows enhanced metafile</td>
</tr>
</tbody>
</table>

You may be familiar with many of these formats. For the sake of completeness, here's a brief description of each:

- **Bmp** The native Windows bitmap file format, also known as the device-independent bitmap (DIB). The DIB was adapted from the OS/2 1.1 bitmap format and was introduced in Windows 3.0. Prior to that time, the format of bitmaps in Windows was based on specific output devices. Although some applications also used the old format for file interchange, it wasn't intended for that purpose.

  The DIB format is mostly defined in the documentation of certain structures used in the Win32 API, specifically `BITMAPFILEHEADER`, `BITMAPINFO`, `BITMAPINFOHEADER`, and their variants. Chapter 15 of my book *Programming Windows*, 5th ed. (Microsoft Press, 1999) has an extensive discussion of the DIB format. DIB files are generally stored without any compression. A little-used RLE compression scheme is defined for some color formats.

- **MemoryBmp** A memory-based DIB, which is a DIB that is not preceded by a `BITMAPFILEHEADER`.

- **Icon** The Windows icon file format, an extension of the Windows DIB format.

- **Gif** Pronounced with a soft G (like "jif"), the Graphics Interchange Format was developed in the late 1980s for use on CompuServe (an early online information service) and remains one of the two most popular graphics formats on the World Wide Web. The GIF file format uses LZW compression. The Gif87a.txt and Gif98a.txt documents that describe the file format can be found in many different locations on the Web. The GIF specification includes a rudimentary (but popular) animation facility.


  As defined in the standard, however, JPEG is *not* a file format. What is commonly referred to as the JPEG file format is more correctly called the JPEG File Interchange Format (JFIF) and is described in the document [http://www.jpeg.org/public/jfif.pdf](http://www.jpeg.org/public/jfif.pdf). JFIF incorporates a lossy JPEG...
compression technique and has become (with GIF) one of the two most popular graphics formats on the Web. (Neither GIF nor JPEG is specifically referred to in the HTML specification, however, so they're really de facto Web standards.)

- **Png** Pronounced "ping," Portable Network Graphics is a lossless format that was developed under the auspices of the World Wide Web Consortium (W3C) as a license-free alternative to GIF. Most modern Web browsers support PNG as well as GIF and JPEG. A good place to begin exploring PNG is [http://www.w3c.org/Graphics/PNG](http://www.w3c.org/Graphics/PNG). If you want to compress nonphotographic images and you'd prefer to avoid GIF, PNG is the best alternative.

- **Tiff** The Tag Image File Format was originally developed by Aldus (creators of the popular PageMaker application) and Microsoft, and the specification is now owned by Adobe. The 121-page TIFF 6.0 specification in PDF format is available at [http://partners.adobe.com/asn/developer/pdfs/in/TIFF6.pdf](http://partners.adobe.com/asn/developer/pdfs/in/TIFF6.pdf).

- **Exif** The Exchangeable image format was developed by the organization now known as Japan Electronic Industry Development Association (JEIDA) for use in digital cameras. The specification is published on the Web site of the International Imaging Industry Association (I3A), formerly the Photographic and Imaging Association, Inc. (PIMA) at [http://www.pima.net/standards/it10/PIMA15740/Exif_2-1.PDF](http://www.pima.net/standards/it10/PIMA15740/Exif_2-1.PDF).

- **Wmf** This is not a bitmap file format! It is a format for a metafile, which is a collection of drawing functions (generally vector drawing functions) stored in binary form. WMF refers to the old Windows metafile format used prior to the 32-bit versions of Windows.

- **Emf** The Windows enhanced metafile format, inaugurated in the 32-bit versions of Windows. Although you can treat metatiles like other images, they're really in a class of their own. I'll cover metatiles in more detail in Chapter 23.

The Data Compression Book, 2nd ed. (New York: M&T Books, 1985) by Mark Nelson and Jean-Loup Gailly is a good source for the history, technology, and programming of data compression techniques.

**Loading and Drawing**

The `Image` class is abstract, which means that you can't instantiate it using a constructor. However, as mentioned previously, the class includes four static methods that let you obtain an `Image` object, and two that let you obtain a `Bitmap` object:

**Image Static Methods (selection)**

```
Image Image.FromFile(string strFilename)
Image Image.FromFile(string strFilename, bool bUseImageColorManagement)
Image Image.FromStream(Stream stream)
Image Image.FromStream(Stream stream, bool bUseImageColorManagement)
Bitmap Image.FromHbitmap(IntPtr hBitmap)
Bitmap Image.FromHbitmap(IntPtr hBitmap, IntPtr hPalette)
```

You probably won't use the last two methods in this table unless you're interfacing with Win32 code. The first two methods, however, are quite powerful and very simple, as you can see here:

```csharp
Image image = Image.FromFile("CuteCat.jpg");
```

One nice feature of this method is that it uses the file's contents rather than the filename extension to determine the file's format. For example, even if CuteCat.jpg were actually a PNG file that you had perversely named, `FromFile` would still work. If the file can't be found or can't be opened, or if something is wrong with the file's contents, `FromFile` throws an exception.

The second two methods use an object of type `Stream` rather than a filename, but files and streams are closely related. `Stream` is an abstract class in the `System.IO` namespace that implements methods such as `Read`, `Write`, and `Seek`. In many cases, a stream is simply an open file. However, a `Stream` object can also represent sequential data that is stored in a block of memory or that travels on a network connection. Appendix A focuses on files and streams.

The Boolean arguments for `FromFile` and `FromStream` deal with color management, a subject that I'm afraid is beyond the scope of this book.
Although these first four static methods are documented as returning an object of type `Image`, if you call `GetType` on the return value, you'll find the return value to be either of type `System.Drawing.Bitmap` or `System.Drawing.Imaging.Metafile`, depending on the type of file (or stream) you've loaded.

For many of the programs in this chapter, I'll be using a file named Apollo11FullColor.jpg that is stored on the companion CD in the Images And Bitmaps directory that contains the projects for this chapter. This image is the famous portrait of astronaut Buzz Aldrin taken by crewmate Neil Armstrong with a Hasselblad camera on the surface of the moon. As the filename implies, the bitmap is a full-color 24-bit-per-pixel image. Also included in the same directory for your experimentation are Apollo11GrayScale.jpg and Apollo11Palette.png, both of which store 8 bits per pixel. The images in all three files are 220 pixels wide and 240 pixels high. The resolution encoded in the JPEG files is 72 dpi.

Just to get started, here's a small program that uses `Image.FromFile` to obtain an `Image` object and the `Graphics` method `DrawImage` to display it.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageFromFile : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new ImageFromFile());
    }
    public ImageFromFile()
    {
        Text = "Image From File";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Image image = Image.FromFile("..\..\..\Apollo11FullColor.jpg");
        grfx.DrawImage(image, 0, 0);
    }
}
```

The argument to `FromFile` indicates the location of the JPEG file relative to the location of the ImageFromFile.exe file. If you've moved things around, the `FromFile` method won't find the file and will throw an exception. The arguments to `DrawImage` indicate the position of the upper left corner of the image relative to the client area. Using the point (0, 0) positions the image on the left and top edges:
You can also click on the client area to print the image. (Notice that I derived this class from the `PrintableForm` class we created in Chapter 5, "Lines, Curves, and Area Fills.") You'll probably be pleased to see that the printed image appears to be normal size and isn't shrunk to the size of a postage stamp, as so often happens when bitmap images are naively printed.

For comparison, you might want to load this same JPEG file into another application (for example, the Imaging program that comes with Windows 2000) and use the application's display options to display the photo at 100 percent. You'll probably find that the ImageFromFile program displays the image somewhat larger than other applications. Why this is so I'll explain soon.

Meanwhile, however, I want to call your attention to two flaws in ImageFromFile that I ignored just so you could see how easy it is to load and display a bitmap file. It's really not as easy as ImageFromFile implies! The first problem I already mentioned: if the image file isn't where FromFile expects it to be, the method will throw an exception. The program should be prepared to catch that exception. The second flaw involves the location of the FromFile call. In such a program, the call really needs to be executed only once, most conveniently in the program's constructor. The program can save the `Image` object in a field and access it from the `OnPaint` method. Here's a better version of the program.

`BetterImageFromFile.cs`

```csharp
// BetterImageFromFile.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class BetterImageFromFile: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new BetterImageFromFile());
    }

    public BetterImageFromFile()
    {
        Image = null;
    }

    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);

        // Draw the image with its current location
        Graphics g = e.Graphics;
        g.DrawImage(image, new Point(10, 10));
    }
}
```

For comparison, you might want to load this same JPEG file into another application (for example, the Imaging program that comes with Windows 2000) and use the application's display options to display the photo at 100 percent. You'll probably find that the ImageFromFile program displays the image somewhat larger than other applications. Why this is so I'll explain soon.

Meanwhile, however, I want to call your attention to two flaws in ImageFromFile that I ignored just so you could see how easy it is to load and display a bitmap file. It's really not as easy as ImageFromFile implies! The first problem I already mentioned: if the image file isn't where FromFile expects it to be, the method will throw an exception. The program should be prepared to catch that exception. The second flaw involves the location of the FromFile call. In such a program, the call really needs to be executed only once, most conveniently in the program's constructor. The program can save the `Image` object in a field and access it from the `OnPaint` method. Here's a better version of the program.

`BetterImageFromFile.cs`

```csharp
// BetterImageFromFile.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class BetterImageFromFile: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new BetterImageFromFile());
    }

    public BetterImageFromFile()
    {
        Text = "Better Image From File";
    }

    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);

        // Draw the image with its current location
        Graphics g = e.Graphics;
        g.DrawImage(image, new Point(10, 10));
    }
}
```
string strFileName = "..\\..\..\..\Apollo11FullColor.jpg";

try
{
    image = Image.FromFile(strFileName);
}
catch
{
    MessageBox.Show("Cannot find file " + strFileName + "!",
        Text, MessageBoxButtons.OK, MessageBoxIcon.Hand);
}
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    if (image == null)
        return;

    grfx.DrawImage(image, 0, 0);
}

For purposes of clarity, however, I generally won't be checking for the existence of the file in the remaining programs in this chapter.

The static Image.FromStream method, demonstrated in this next program, is useful if you have access to an open file or if you obtain a stream from a source other than a file system. For example, you can use the FromStream method to load an image from the Internet. Here's a program that accesses the NASA Web site to load the original file that I cropped (and made a couple other changes to) to create the Apollo11 files included with this chapter.

ImageFromWeb.cs
//-------------------------------
// ImageFromWeb.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Net;
using System.Windows.Forms;

class ImageFromWeb: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new ImageFromWeb());
    
}
public ImageFromWeb()
{
    Text = "Image From Web";

    WebRequest webreq = WebRequest.Create(strUrl);
   WebResponse webres = webreq.GetResponse();
    Stream stream = webres.GetResponseStream();

    image = Image.FromStream(stream);
    stream.Close();
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.DrawImage(image, 0, 0);
}

The statements using the WebRequest andWebResponse classes represent the standard approach to downloading Web files. In this program, the GetResponseStream method ofWebResponse obtains a readable stream of the JPEG file. At that point, you can just pass that stream to the Image.FromStream method:

image = Image.FromStream(stream);

The ImageFromWeb program is missing a few features that should be standard in any program that downloads files from the Web. It probably should include a progress bar (easily implemented as an object of type ProgressBar), and the stream-reading code should probably be in a second thread of execution.
The Image class contains several properties that provide information about the object. First are three properties that indicate the size of the image in pixels:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Size</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Height</td>
<td>get</td>
</tr>
</tbody>
</table>

The Width and Height properties are consistent with the Size property. Use whichever is convenient.

Most modern bitmap formats include some indication of the resolution of the image in dots per inch or an equivalent. Such a resolution might not make much sense for some images, including the image I just displayed. You can display such an image larger or smaller and it's still the same. But for some bitmaps—perhaps ones in which the image is supposed to match the size of the object it portrays—some indication of the physical size of the image is helpful.

Where does the resolution come from? Usually from the program that originally creates the bitmap. For example, if you scan an image on a scanner at 300 dpi, the scanning software usually sets the resolution of the resultant image at 300 dpi. When you create an image in a paint application, it usually sets the resolution to the screen resolution under the assumption that you've made the image the appropriate size for your screen.

Additional properties of Image let you obtain the horizontal and vertical resolution of the Image object and the resultant metrical size:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>HorizontalResolution</td>
<td>get</td>
<td>In dots per inch</td>
</tr>
<tr>
<td>float</td>
<td>VerticalResolution</td>
<td>get</td>
<td>In dots per inch</td>
</tr>
<tr>
<td>SizeF</td>
<td>PhysicalDimension</td>
<td>get</td>
<td>In hundredths of millimeters</td>
</tr>
</tbody>
</table>

If the image doesn't have any resolution information, the HorizontalResolution and VerticalResolution properties return the resolution of the video display. You might want to ignore PhysicalDimension (particularly because it wasn't working right in early versions of Windows Forms) and calculate a metrical size of the image yourself. For example, the following statements calculate the size of the image in inches:

```csharp
float cxInches = image.Width / image.HorizontalResolution;
float cyInches = image.Height / image.VerticalResolution;
```

The programs shown so far in this chapter used the following version of the DrawImage function to display the image:

```csharp
gfx.DrawImage(image, x, y)
```

This method sizes the image based on its metrical dimension! That's why the programs shown earlier in this chapter display the image at a little different size than other Windows applications. We'll examine the image rendering methods in more detail later in this chapter.

Another property of the Image class indicates the image's pixel format. The pixel format indicates the color depth and how the pixels correspond to colors:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PixelFormat</td>
<td>PixelFormat</td>
<td>get</td>
</tr>
</tbody>
</table>

This property will return one of the following members of the PixelFormat enumeration, defined in the System.Drawing.Imaging namespace:
The number after the word Format indicates the number of bits per pixel: 1, 4, 8, 16, 32, 48, or 64. Those formats that have 1, 4, or 8 bits per pixel are indexed, which means that the pixel values are indices into a color palette. The formats containing the letters Rgb store red, green, and blue values for each pixel. The Argb formats also include an alpha channel for transparency. The PArgb formats contain red, green, and blue values that have been premultiplied by the alpha value. Just off hand, the numerical values of the enumeration members might appear to be random, but if you look more closely, you’ll find some definite patterns. Take a look at the rightmost two hexadecimal digits. Each value is unique, ranging from 0x00 (the ominous Undefined or DontCare value) to 0x0E.

The next two rightmost digits indicate the number of bits per pixel: 0x01, 0x04, 0x08, 0x10, 0x18, 0x20, 0x30, or 0x40. The other bits are flags. The following PixelFormat enumeration values include Max, which indicates that the number of formats (including Undefined or DontCare) is 15, plus the values that explain the meaning of the flags:

The Image class also contains several static methods that let you extract most of this information without digging into the bits:
int GetPixelFormatSize(PixelFormat pf)
bool IsAlphaPixelFormat(PixelFormat pf)
bool IsCanonicalPixelFormat(PixelFormat pf)
bool IsExtendedPixelFormat(PixelFormat pf)

The first of these methods returns the number of bits per pixel.

If the image is indexed, which you can determine by performing a bitwise AND with the PixelFormat property and PixelFormat.Indexed, the image has a color palette. You can obtain that palette from the Palette property:

<table>
<thead>
<tr>
<th>Image Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>ColorPalette</td>
</tr>
</tbody>
</table>

This property is quite peculiar. You'll notice that it's both readable and writable. However, the ColorPalette class itself (which is defined in the System.Drawing.Imaging namespace) is sealed, which means that you can't subclass it, and it has no public constructors, which means that you can't instantiate it. There is no way in all of .NET to obtain a ColorPalette object except as this property of the Image class.

ColorPalette itself has just two, read-only properties:

<table>
<thead>
<tr>
<th>ColorPalette Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Color[]</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

The Entries property returns the array of colors in the image's color palette.

Another property of the Image class indicates the file format:

<table>
<thead>
<tr>
<th>Image Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>ImageFormat</td>
</tr>
</tbody>
</table>

I've already mentioned the ImageFormat class. That's the class that contains a static property for each of the supported bitmap file formats as shown in the table earlier in this chapter (page 482).

The RawFormat property of the Image class is a bit difficult to use, however. You have to use it in conjunction with the only instance property (that is, the only nonstatic property) of the ImageFormat class, which returns a globally unique identifier (GUID) for an ImageFormat object:

<table>
<thead>
<tr>
<th>ImageFormat Instance Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Guid</td>
</tr>
</tbody>
</table>

Your program can test only whether a particular Image object is a particular ImageFormat type. For example, if you have an Image object named image that was loaded from a JPEG file, the expression


returns false and

returns `true`. The expression
```
image.RawFormat.ToString()
```
returns the string
```
[ImageFormat: b96b3cae-0728-11d3-9d7b-0000f81ef32e]
```
while the expression
```
ImageFormat.Jpeg.ToString()
```
returns the string
```
Jpeg
```
You know which one to display to a user, right?

I'll cover some other `Image` properties and methods later in this chapter.

**Rendering the Image**

As the sample programs shown so far in this chapter illustrate, the method of the `Graphics` class that displays images is called `DrawImage`, a method that comes in a whopping 30 versions for much flexibility. Another image-drawing method, named `DrawImageUnscaled`, is also available, but it provides no additional functionality over `DrawImage`.

As with the display of text, the display of images involves dealing with an object that already has a specific size. Just as a text string associated with a certain font has a size, a bitmap image also has a size, or rather two sizes: a pixel size and a metrical size. Displaying images in their metrical size (which the simplest versions of `DrawImage` do) is helpful when you're attempting to treat images in a device-independent manner. However, if you're otherwise drawing in units of pixels, you need to use a bit of math to anticipate the pixel size of such a rendered image. (I'll show you that math shortly.) At times—particularly when integrating images with controls—you'll want to display an image in its pixel size. `DrawImage` doesn't use the image's pixel size automatically, but it's easy to persuade the method to draw an image in its pixel size.

Traditionally, bitmaps have been resistant to transforms such as rotation. In Windows Forms and GDI+, the display of bitmaps is always affected by the world transform in much the same way as text.

The first argument to all the versions of `DrawImage` is an object of type `Image`. At the very least, the method also always includes a coordinate point. This point is in the form of two integers, two floats, a `Point`, a `PointF`, a `Rectangle`, or a `RectangleF` and indicates in world coordinates where the upper left corner of the image appears.

These four `DrawImage` methods size the image based on its metrical dimensions:

**Graphics DrawImage Methods (selection)**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void DrawImage(Image image, int x, int y)</td>
</tr>
<tr>
<td>void DrawImage(Image image, float x, float y)</td>
</tr>
<tr>
<td>void DrawImage(Image image, Point pt)</td>
</tr>
<tr>
<td>void DrawImage(Image image, PointF ptf)</td>
</tr>
</tbody>
</table>

The size of the image is unaffected by any page transform but is affected by the world transform. Using these four methods of `DrawImage` is analogous to calling `DrawString` with a `Font` object created with a metrical size. The resultant image is the same metrical size on both the video display and the printer.

For example, the Apollo11 images are 220 pixels wide and 240 pixels high, and they have a resolution of 72 dpi. Thus, when displayed by the versions of `DrawImage` I've talked about so far, the width of the images is about 3 inches and the height is 3-1/3 inches.
The JPEG from the NASA Web site that the ImageFromWeb program accesses is 640 × 480 pixels in size but has no embedded resolution information. In such a case, the resolution of the image is assumed to be your screen resolution, which is probably 96 or 120 dpi.

At times, you need to anticipate how large an image will be when it's displayed. For example, you might need to center an image within a rectangle. Because the four versions of DrawImage we've seen so far draw the image in its metrical size, centering that image within the client area requires a bit of work, as illustrated in the following program.

```
CenterImage.cs
//------------------------------------------
// CenterImage.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CenterImage: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new CenterImage());
    }
    public CenterImage()
    {
        Text = "Center Image";

        image = Image.FromFile("..\..\..\..\Apollo11FullColor.jpg");
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.PageUnit = GraphicsUnit.Pixel;
        grfx.PageScale = 1;

        RectangleF rectf = grfx.VisibleClipBounds;

        float cxImage = grfx.DpiX * image.Width / 
                        image.HorizontalResolution;

        float cyImage = grfx.DpiY * image.Height / 
                        image.VerticalResolution;

        grfx.DrawImage(image, (rectf.Width - cxImage) / 2,
                        (rectf.Height - cyImage) / 2);
    }
}
```
The cxImage and cyImage values are in units of pixels: dividing the pixel width and height of the image by the horizontal and vertical resolution provides the dimension of the image in inches, and then multiplying that by the DpiX and DpiY properties yields the dimension in device pixels of the displayed image.

If you were dealing only with the video display, you could then subtract cxImage and cyImage from the width and height of the client area and divide by 2. However, that method won't work on the printer. Instead, at the beginning of the DoPage method, I switch to pixel coordinates and use VisibleClipBounds to obtain the dimension of the output device in pixels. The cxImage and cyImage values are then subtracted from the pixel width and height of the device and divided by 2.

Fitting to a Rectangle

The following four DrawImage methods specify a rectangular destination for the image. The rectangle is in world coordinates:

**Graphics DrawImage Methods (selection)**

```csharp
void DrawImage(Image image, int x, int y, int cx, int cy)
void DrawImage(Image image, float x, float y, float cx, float cy)
void DrawImage(Image image, Rectangle rect)
void DrawImage(Image image, RectangleF rectf)
```

These methods scale the image to the size of the rectangle, either stretching or compressing it to fit. One common use of these methods is to display an image in its pixel size rather than its metrical size. If page units are pixels, simply call

```csharp
grfx.DrawImage(image, x, y, image.Width, image.Height);
```

The following program displays an image in its pixel dimensions centered within the client area (or printer page).

**CenterPixelSizeImage.cs**

```csharp
//---------------------------------------------------
// CenterPixelSizeImage.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CenterPixelSizeImage: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new CenterPixelSizeImage());
    }
    public CenterPixelSizeImage()
    {
        Text = "Center Pixel-Size Image";

        image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
    }
}
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.DrawImage(image, (cx - image.Width) / 2, 
                   (cy - image.Height) / 2, 
                   image.Width, image.Height);
}

Because your video resolution is most likely greater than 72 dpi, this image is smaller than the one drawn by DrawImage:

On the printer, which has a default page transform that makes it appear to be a 100-dpi device, this version of the DrawImage method will render the 220 × 240 pixel bitmap as 2.2 × 2.4 inches. If you set page units to pixels in the DoPage method, the printed image will be much smaller, probably resulting in the postage stamp effect commonly encountered in less sophisticated graphics programming environments.

The following program loads an image and scales it to the entire size of the client area (or printable area of the printer page).

ImageScaleToRectangle.cs
//----------------------------------------------------
// ImageScaleToRectangle.cs © 2001 by Charles Petzold
//------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageScaleToRectangle: PrintableForm
{
    Image image;

    public new static void Main()
As you make the client area much wider than it is tall, or much taller than it is wide, the image is distorted accordingly:

If you really do need to scale an image to the size of a rectangle, this effect is probably not what you had in mind. You probably want to scale the image isotropically, which means equally in both directions. Here's a program that scales a rectangle more intelligently.

ImageScaleIsotropic.cs
//--------------------------------------------------
// ImageScaleIsotropic.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageScaleIsotropic: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new ImageScaleIsotropic());
    }

    public ImageScaleIsotropic()
    {
        Text = "Image Scale Isotropic";

        image = Image.FromFile("..\..\..\..\\\Apollo11FullColor.jpg");
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawImage(image, 0, 0, cx, cy);
    }
}
The `ScaleImageIsotropically` method will work in all cases except when the horizontal and vertical resolutions of the device are different (as is the case with some printers) and when the `PageUnit` is `GraphicsUnit.Pixel` (which is not the default case for printers).

The method begins by calculating a `SizeF` structure that indicates the size of the `Image` object in inches. (This step wouldn't be necessary if the horizontal and vertical resolution of the image were the same.) Then a factor is calculated that is the minimum of the destination rectangle width and height divided by the image width and height. This `fScale` number is the factor that must be applied to the image size to isotropically scale it to the size of the destination rectangle. The method then calculates the origin of this rectangle and passes all the information to `DrawImage`. Here's the image:

![Image](image.png)

The rectangle-destination versions of the `DrawImage` method can do additional tricks beyond just stretching an image. If you specify a negative width, the image is flipped around the vertical axis—it's a mirror image. A negative height flips the image around the horizontal axis and shows it upside down. In all cases, the upper left corner of the original unflipped image is always positioned at the `Point` or `PointF` portion of the rectangle you specify in the drawing method.
Let's look at a program that draws four images, some with negative widths and heights. In all four cases, the second and third arguments to DrawImage indicate the center of the client area.

**ImageReflection.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageReflection : PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new ImageReflection());
    }

    public ImageReflection()
    {
        Text = "Image Reflection";
        image = Image.FromFile("..\..\..\\\\Apollo11FullColor.jpg");
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        int cxImage = image.Width;
        int cyImage = image.Height;

        grfx.DrawImage(image, cx / 2, cy / 2, cxImage, cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2, -cxImage, cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2, cxImage, -cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2, -cxImage, -cyImage);
    }
}
```

And here's the result showing the four images:
Notice that the program sizes the image based on its pixel dimension.

**Rotate and Shear**

You can distort the image even more using the following two methods. These methods effectively translate, scale, shear, or rotate an image into a parallelogram.

**Graphics DrawImage Methods (selection)**

```csharp
void DrawImage(Image image, Point[] apt)
void DrawImage(Image image, PointF[] aptf)
```

The array argument must contain exactly three points. These points indicate the destination in world coordinates of three corners of the image:

- `apt[0]` = destination of upper left corner of image
- `apt[1]` = destination of upper right corner of image
- `apt[2]` = destination of lower left corner of image

Because the resulting image is a parallelogram, the destination of the lower right corner of the image is implied.

Here's a program that sets these three points to the center of the top side of the client area, the center of the right side of the client area, and the center of the left side of the client area.

**ImageAtPoints.cs**

```csharp
//--------------------------------------------
// ImageAtPoints.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageAtPoints: PrintableForm
{
    Image image;
```
public new static void Main()
{
    Application.Run(new ImageAtPoints());
}

public ImageAtPoints()
{
    Text = "Image At Points";

    image = Image.FromFile("..\..\..\..\Apollo11FullColor.jpg");
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.DrawImage(image, new Point[] { new Point(cx / 2, 0),
                                           new Point(cx, cy / 2),
                                           new Point(0, cy / 2)});
}

And here's the image:

This isn't the only way to rotate or shear bitmap images. You can also use the normal world transform.

Displaying Part of the Image

If you've kept count, you'll know that so far I've covered only 10 of the 30 versions of the DrawImage method. All the remaining methods let you specify a rectangular subsection of the bitmap to display. You specify this subsection in pixels relative to the upper left corner of the image. For an Image object named image, the rectangle

new Rectangle(0, 0, image.Width, image.Height)
Here are two versions of `DrawImage` that specify the destination as a point in world coordinates, a rectangular source specifying a subset of the image, and a `GraphicsUnit` argument:

```
void DrawImage(Image image, int xDst, int yDst,
               Rectangle rectSrc, GraphicsUnit gu)
void DrawImage(Image image, float xDst, float yDst,
               RectangleF rectfSrc, GraphicsUnit gu)
```

The concept here is simpler than the bizarre definitions of these methods would seem to imply. First, you always specify the source rectangle in units of pixels. (Thus, the version of `DrawImage` defined with a `RectangleF` structure rather than a `Rectangle` structure makes no sense.) Second, the `GraphicsUnit` argument must be `GraphicsUnit.Pixel`. I happen to know that the coordinates of Buzz Aldrin's helmet in the image I've been using can be expressed approximately by the rectangle

```
new Rectangle(95, 0, 50, 55)
```

Let's take a look at a program that displays just this portion of the image.

```
// PartialImage.cs
//---------------------------------
//-- PartialImage.cs © 2001 by Charles Petzold
//---------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PartialImage : PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new PartialImage());
    }
    public PartialImage()
    {
        Text = "Partial Image";

        image = Image.FromFile("..\..\..\..\Apollo11FullColor.jpg");
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Rectangle rect = new Rectangle(95, 0, 50, 55);
    }
}
```
grfx.DrawImage(image, 0, 0, rect, GraphicsUnit.Pixel);
}
}

How large is the rendered image? Because the resolution of the file is 72 dpi, the image is drawn 50/72 inch wide and 55/72 inch high:

There are also four methods that let you specify both a source rectangle and a destination rectangle:

**Graphics DrawImage Methods (selection)**

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Argument List</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>void DrawImage(Image image, Rectangle rectDst, int xSrc, int ySrc, int cxSrc, int cySrc, GraphicsUnit gu, *)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void DrawImage(Image image, Rectangle rectDst, Rectangle rectSrc, GraphicsUnit gu)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void DrawImage(Image image, Rectangle rectDst, float x, float y, float cx, float cy, GraphicsUnit gu, *)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void DrawImage(Image image, RectangleF rectfDst, RectangleF rectfSrc, GraphicsUnit gu)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In these methods, the source rectangle is in pixels and the destination rectangle is in world coordinates. The `GraphicsUnit` argument must be `GraphicsUnit.Pixel`.

Notice in this table that I've put an asterisk at the end of the argument list for two of the methods. The asterisk means that these methods can also have three optional arguments: an `ImageAttribute` object, a callback function to abort drawing an image, and data to pass to the callback function; thus, each of the two methods with the asterisks has three additional versions. (Unfortunately, I won't be able to discuss these additional versions of `DrawImage` in this book.)
Here's a program that displays Aldrin's helmet (the source rectangle) the size of the client area or the printer page (the destination rectangle).

**PartialImageStretch.cs**

```
//---------------------------------
//-- PartialImageStretch.cs © 2001 by Charles Petzold
//---------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PartialImageStretch: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new PartialImageStretch());
    }
    public PartialImageStretch()
    {
        Text = "Partial Image Stretch";

        image = Image.FromFile("..\..\..\..\..\Apollo11FullColor.jpg");
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Rectangle rectSrc = new Rectangle(95, 5, 50, 55);
        Rectangle rectDst = new Rectangle(0, 0, cx, cy);

        grfx.DrawImage(image, rectDst, rectSrc, GraphicsUnit.Pixel);
    }
}
```

Here's the image stretched to the client area:
If the destination rectangle isn't the same aspect ratio as the source rectangle, the image will be distorted. But you can easily adapt the `ScaleImageIsotropically` method I showed earlier (on page 501) to partial images. To display the image in its pixel size, use the same width and height in the destination rectangle as specified in the source rectangle.

When you blow up smaller images like this and print them, you might expect to see jaggies. However, GDI+ performs an interpolation of the image pixels to smooth out the image.

To complete the collection of `DrawImage` methods, you can also display the partial image using an array of three points. The asterisk indicates that each of these methods is actually four methods.

**Graphics DrawImage Methods (selection)**

```csharp
void DrawImage(Image image, Point[] aptDst,
               Rectangle rectSrc, GraphicsUnit gu, *)
void DrawImage(Image image, PointF[] aptfDst
               RectangleF rectfSrc, GraphicsUnit gu, *)
```

Again, the source rectangle is in pixels, the destination points are in world coordinates, and the `GraphicsUnit` argument must be `GraphicsUnit.Pixel`. Here's a sample program.

**PartialImageRotate.cs**

```csharp
//-------------------------------------------------
// PartialImageRotate.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PartialImageRotate : PrintableForm
```
And now, if you've been keeping count, we're finished with the 30 DrawImage methods.

**Drawing on the Image**

We've been drawing a bitmapped image on the video display and the printer. It's also possible to draw on an image. If you think about it, when Windows draws on the video display, it's really drawing on a big bitmap stored in memory on the video display adapter. Many printers, also, base their output on the contents of memory organized like a bitmap. So it makes sense that a Windows program should be able to draw on any bitmap by using the same graphics output functions you use on the video display and the printer.

To draw on an image, you need to obtain a Graphics object that refers to the image. You get that Graphics object from a static method of the Graphics class:

```
Graphics Graphics.FromImage(Image image)
```

For example, here's a statement that obtains a Graphics object named grfxImage based on an Image object named image:

```
Graphics grfxImage = Graphics.FromImage(image);
```

When you're finished with this Graphics object, call the Dispose method to get rid of it.

The Graphics.FromImage method won't work with every image format. The method will not work and will throw an exception if the PixelFormat property of the image is one of the following PixelFormat

```
PixelFormat
```
members: Format1bppIndexed, Format4bppIndexed, Format8bppIndexed, Format16bppGrayScale, or Format16Argb1555. This restriction may make sense to you. Suppose you obtain a Graphics object based on an indexed image and then try to draw on the image with a color that's not in the image's palette table. The problem is similar if you could obtain a Graphics object for a gray-scaled image format or the 1-bit transparency format.

Neither can you obtain a Graphics object for an image loaded from an old Windows metafile (WMF) or a Windows enhanced metafile (EMF). In Chapter 23, I'll discuss some other ways to draw on metafiles.

You can use the Graphics object you obtain from Graphics.FromImage in the same way you use a Graphics object for the video display or the printer. If you check the DpiX and DpiY properties for the image-based Graphics object, you'll find that they are equal to the HorizontalResolution and VerticalResolution properties of the Image object. The default page transform is a PageUnit of GraphicsUnit.Display and a PageScale of 1, which for images is the same as GraphicsUnit.Pixels. By default, the VisibleClipBounds property of this Graphics object is equal to the width and height of the image in pixels.

You can set a different page transform for the Graphics object. If you change the PageUnit and PageScale properties, the VisibleClipBounds property indicates the dimensions of the image in page units. For example, if you set

```csharp
grfxImage.PageUnit = GraphicsUnit.Inch;
grfxImage.PageScale = 1;
```

VisibleClipBounds indicates the size of the image in inches, which you could also calculate by dividing the pixel size of the image by the HorizontalResolution and VerticalResolution properties.

In a few moments, I'll be drawing text on the surface of a bitmap. So the questions naturally arise, What font should I use? Is the default Font property of the form satisfactory?

The default Font property for a form is an 8-point font. If you draw text on a bitmap using this font and then display the image in its metrical size (rather than pixel size), the text will be the same size as if you drew the text directly on the client area. However, if the resolution of the image is less than the resolution of the video display, the text certainly won't look exactly the same. The text on the image will be coarser than the text displayed directly on the window.

Consider this example. The sample bitmaps I've been using in this chapter are 72 dpi—1 pixel per point. That means that an 8-point font at this resolution is about 8 pixels in height. Using the form's default Font property, if you call

```csharp
Font.GetHeight(72)
```

or, using the Graphics object created from one of the bitmaps I've been using in this chapter, if you call

```csharp
Font.GetHeight(grfxImage)
```

you'll get 8.83 pixels for line spacing, confirming a character height of about 8 pixels.

Now, 8 pixels isn't a whole lot of space to draw well-rounded font characters. But that will be the font height DrawString will be limited to if you use the form's default Font property on a 72-dpi image. If the resolution of your video display is higher than 72 dpi and you display that image in its pixel size, the text on the image will appear very small. If you display the image in its metrical size, the image and the text will be larger. On a 96-dpi display, the image (and hence the text in the image) will be increased by a factor of 1-1/3. On a 120-dpi display, the image will be increased by a factor of 1-2/3. So, those 8 pixels are stretched to the normal size of the default Font.

Let's take a look. Here's the program.

```
DrawOnImage.cs
```

```
//------------------------------------------
// DrawOnImage.cs © 2001 by Charles Petzold
//------------------------------------------
```
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawOnImage : PrintableForm
{
    Image image;
    string str = "Apollo11";

    public new static void Main()
    {
        Application.Run(new DrawOnImage());
    }

    public DrawOnImage()
    {
        Text = "Draw on Image";
        image = Image.FromFile("..\..\..\..\..\Apollo11FullColor.jpg");

        Graphics grfxImage = Graphics.FromImage(image);

        grfxImage.PageUnit = GraphicsUnit.Inch;
        grfxImage.PageScale = 1;

        SizeF sizef = grfxImage.MeasureString(str, Font);

        grfxImage.DrawString(str, Font, Brushes.White,
                             grfxImage.VisibleClipBounds.Width - sizef.Width, 0);

        grfxImage.Dispose();
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.PageUnit = GraphicsUnit.Pixel;
        grfx.DrawImage(image, 0, 0);
        grfx.DrawString(str, Font, new SolidBrush(clr),
                         grfx.DpiX * image.Width / image.HorizontalResolution, 0);
    }
}

Just for fun, I changed the page transform of the Graphics object associated with the image to draw in units of inches. I call MeasureString to obtain the dimensions of the string, and I set the coordinates of DrawString so that the string appears in the upper right corner of the image. The DoPage method calls DrawImage to draw the image and then—for comparison—the method also calls DrawString to display the same text string on the right side of the image. (Setting page units to pixels is necessary so that the text is positioned correctly on the printer.) You'll notice that the text on
the image is considerably coarser and somewhat distorted because the image has been stretched
on the display:

Is there anything you can do about this distortion problem? The problem arises from using a font that
has only a 9-pixel height. You can solve the problem only if you use a font with a larger pixel size.
That means using images with resolutions greater than the screen resolution or inflating the font size
so that it's at least 12 pixels or so. The latter approach will make the text look larger, but at least it
will be readable.

If you'll be displaying the image in its pixel size, you need to take a different approach. To make text
on the image the same size as normal text displayed on the client area, you need to scale the font
based on the ratio of the screen resolution to the image resolution. Here's an example.

```
DrawOnPixelSizeImage.cs
//---------------------------------------------------
// DrawOnPixelSizeImage.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawOnPixelSizeImage: PrintableForm
{
    Image image;
    string str = "Apollo11";

    public new static void Main()
    {
        Application.Run(new DrawOnPixelSizeImage());
    }
    public DrawOnPixelSizeImage()
    {
        Text = "Draw on Pixel-Size Image";
        image = Image.FromFile("..\..\..\..\Apollo11FullColor.jpg");

        Graphics grfxImage = Graphics.FromImage(image);
        Graphics grfxScreen = CreateGraphics();

        Font font = new Font(Font.FontFamily,
            grfxScreen.DpiY / grfxImage.DpiY * Font.SizeInPoints);
```
The constructor creates a new Font object (named font) that is the same as the default font but with a size in points scaled by the ratio of the screen resolution to the image resolution. This is the font I use in the calls to MeasureString and DrawString. Because I haven’t set a nondefault page transform, I can use the pixel width of the image in the DrawString call to position the string. The font displayed on the image is visually the same size as the font displayed on the client area:

The two fonts now look identical, even though they are different point sizes. They appear to be the same size because they’re displayed on two different surfaces (the image and the client area) with different display resolutions. Because I sized the font used on the image based on the screen resolution, the two fonts will differ on the printer by the extent to which the 100-dpi virtual resolution of the printer differs from the screen resolution.

I didn’t mention my use of Brushes.White in the DrawString call to draw on the image. It's exactly what's appropriate, of course, given the black background, but suggests also that you really need to know what's on an existing image before you start drawing on it!

You can also draw on blank bitmaps that you create specifically for that purpose. I'll show you how to do that shortly.

More on the Image Class
The `Image` class has several additional methods that let you save or manipulate the image in limited ways. The first version of `Save` in the following list uses the filename extension to determine the file format, but you can be more explicit in the next two versions:

**Image Save Methods (selection)**

```csharp
void Save(string strFilename)
void Save(string strFilename, ImageFormat if)
void Save(Stream stream, ImageFormat if)
```

You can't use `Save` on any `Image` object loaded from a metafile or a memory bitmap. Nor can you save an image in a metafile or a memory bitmap format.

The following two methods can resize an image and rotate and flip it in certain fixed ways, respectively:

**Image Methods (selection)**

```csharp
Image GetThumbnailImage(int cx, int cy,
                        Image.GetThumbnailImageAbort gtia,
                        IntPtr pData
                        void RotateFlip(RotateFliptype rft);
```

The `GetThumbnailImage` method is intended to be used to create a thumbnail of an image, which is a smaller version of the image that an application can use to convey the contents of the image while saving time and space. However, `GetThumbnailImage` is actually a general-purpose image-resizing function. You can make the image larger as well as smaller. The last two arguments are used to specify a callback function, but you can set them to `null` and 0 and the method will work just fine without them. Here's a program that creates a thumbnail designed to fit in a 64-pixel-square space.

**Thumbnail.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class Thumbnail : PrintableForm
{
    const int iSquare = 64;
    Image imageThumbnail;

    public new static void Main()
    {
        Application.Run(new Thumbnail());
    }

    public Thumbnail()
    {
```
Text = "Thumbnail";

Image image = Image.FromFile("..\..\..\..\..\Apollo11FullColor.jpg");

int cxThumbnail, cyThumbnail;

if (image.Width > image.Height)
{
    cxThumbnail = iSquare;
    cyThumbnail = iSquare * image.Height / image.Width;
}
else
{
    cyThumbnail = iSquare;
    cxThumbnail = iSquare * image.Width / image.Height;
}

imageThumbnail = image.GetThumbnailImage(cxThumbnail, cyThumbnail,
                                           null, (IntPtr) 0);

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    for (int y = 0; y < cy; y += iSquare)
    for (int x = 0; x < cx; x += iSquare)
        grfx.DrawImage(imageThumbnail,
                        x + (iSquare - imageThumbnail.Width) / 2,
                        y + (iSquare - imageThumbnail.Height) / 2,
                        imageThumbnail.Width,
                        imageThumbnail.Height);
}

The program handles the DoPage method by filling up its client area (or the printer page) with the thumbnail image:
While the `GetThumbnailImage` method creates a new image, the `RotateFlip` method alters the existing image. The single argument is a member of the `RotateFlipType` enumeration:

### `RotateFlipType` Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RotateNoneFlipNone</code></td>
<td>0</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate180FlipXY</code></td>
<td>1</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate90FlipNone</code></td>
<td>2</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate270FlipXY</code></td>
<td>3</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate180FlipNone</code></td>
<td>4</td>
<td>hello</td>
</tr>
<tr>
<td><code>RotateNoneFlipX</code></td>
<td>5</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate90FlipX</code></td>
<td>6</td>
<td>Hello</td>
</tr>
<tr>
<td><code>Rotate270FlipX</code></td>
<td>7</td>
<td>Hello</td>
</tr>
</tbody>
</table>

Although the enumeration has 16 members, there are only 8 unique effects on the image. For 4 of these effects, the `Image` object gets its `Width` and `Height` properties switched around.

If you need to rotate or flip an image but you still want to retain the original unflipped unrotated image, you can first make a copy of the original `Image` object by using the `Clone` method:

```csharp
Image imageCopy = (Image) image.Clone();
```

### The Bitmap Class

So far, everything I've been discussing uses objects of type `Image`. As I mentioned at the beginning of this chapter, the `System.Drawing` namespace also includes a class named `Bitmap` that inherits from `Image`. All the `Image` properties apply to `Bitmap` as well. Anything you can do with an `Image`
object you can also do with a Bitmap object. Plus more, of course. The Bitmap class allows you to get down and dirty with the bitmap bits.

The Image class has no constructors; the Bitmap class has 12 constructors. These first constructors load a Bitmap object from a file, a stream, or a resource:

**Bitmap Constructors (selection)**

- Bitmap(string strFilename)
- Bitmap(string strFilename, bool bUseImageColorManagement)
- Bitmap(Stream stream)
- Bitmap(Stream stream, bool bUseImageColorManagement)
- Bitmap(Type type, string strResource)

The first four constructors basically duplicate the static FromFile and FromStream methods implemented in Image. The last loads a Bitmap object as a resource, which is usually embedded in the .exe file of the application. I'll explain resources later in this chapter.

Next is a collection of constructors that create new Bitmap objects based on an existing Image object:

**Bitmap Constructors (selection)**

- Bitmap(Image image)
- Bitmap(Image image, Size size)
- Bitmap(Image image, int cx, int cy)

Although the first argument of these constructors is defined as an Image, it can also be another Bitmap object. The first constructor is similar to the Clone method of the Image class; it creates a new Bitmap object that is identical to the first. The second and third constructors are similar to the GetThumbnailImage method; the image is resized. In all cases, the new bitmap inherits the pixel format of the existing bitmap. In all cases, the resolution of the bitmap is set to the resolution of the video display.

The final four constructors have no equivalents in the Image class. These constructors let you create brand new Bitmap objects with blank images:

**Bitmap Constructors (selection)**

- Bitmap(int cx, int cy)
- Bitmap(int cx, int cy, PixelFormat pf)
- Bitmap(int cx, int cy, Graphics grfx)
- Bitmap(int cx, int cy, int cxRowBytes, PixelFormat pf, IntPtr pBytes)

The first three constructors initialize the pixels to 0. A pixel value of 0 has a different meaning for different types of bitmaps. For RGB bitmaps, 0 means black. For ARGB bitmaps, 0 means transparent. The fourth constructor also allows you to pass a pointer to an array of bytes that initializes the bitmap image.

The first constructor in the table creates a Bitmap object of the specified size with a pixel format of PixelFormat.Format32bppArgb. That's 32 bits per pixel with an alpha channel for transparency. The horizontal and vertical resolution are set to the resolution of the video display. The second
constructor lets you specify a PixelFormat member if you want something other than Format32bppArgb.

The third constructor lets you specify a Graphics object. Regardless of whether you specify a Graphics object associated with the video display or the printer, and regardless of whether or not your printer is capable of color, the constructor always creates a Bitmap with a pixel format of PixelFormat.Format32bppPArgb. Notice that this pixel format implies a premultiplied alpha channel. The really important implication of creating a bitmap based on a Graphics object is that the HorizontalResolution and VerticalResolution properties of the Bitmap object are set to the DpiX and DpiY properties of the Graphics object. And that doesn't mean 100 dpi for the printer! That means 300, 600, 720, or something higher.

For example, suppose your printer has a resolution of 600 dpi. You want to create a bitmap based on the printer resolution. And why do you want to do this? Because if you'll eventually be printing the bitmap, you want anything you draw on this bitmap (including text) to be as fine and as rounded as the printer resolution will allow. But keep in mind that the bitmap size must be compatible with the resolution. A 2-inch-square 600-dpi bitmap will require widths and heights of 1200 pixels and will consume over 5 megabytes of memory. And be sure to use the metrical size when you display such a bitmap on the screen and the printer! (Don't worry: I'll have an example shortly.)

At any rate, if you prefer to create a bitmap with a resolution that matches neither the screen nor the printer, the Bitmap class provides a method that allows you to change the resolution of a bitmap you've loaded or created:

**Bitmap Methods (selection)**

```csharp
void SetResolution(float xDpi, float yDpi)
```

How do you get stuff on the surface of the bitmap? There are three approaches:

- You can create a Graphics object for the bitmap and draw on the bitmap as if it were any other graphics device. I described this approach earlier in the chapter. Remember that you can't create a Graphics object for every possible pixel format!
- You can use the SetPixel and GetPixel methods of the Bitmap class to set (or get) the color of individual pixels in the bitmap.
- You can use the Bitmap class methods LockBits and UnlockBits to get access to the actual bitmap data.

I won't be able to demonstrate the second and third approaches in this book.

**Hello World with a Bitmap**

This HelloWorldBitmap program creates a bitmap and draws on the bitmap a 72-point version of the programmer's universal mantra. It then displays that bitmap on the client area and (optionally) on your printer page.

**HelloWorldBitmap.cs**

```csharp
//-----------------------------
// HelloWorldBitmap.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloWorldBitmap: PrintableForm
{
    const float fResolution = 300;
    Bitmap bitmap;
```
public new static void Main()
{
    Application.Run(new HelloWorldBitmap());
}
public HelloWorldBitmap()
{
    Text = "Hello, World!";
    bitmap = new Bitmap(1, 1);
    bitmap.SetResolution(fResolution, fResolution);

    Graphics grfx = Graphics.FromImage(bitmap);
    Font font = new Font("Times New Roman", 72);
    Size size = grfx.MeasureString(Text, font).ToSize();

    bitmap = new Bitmap(bitmap, size);
    bitmap.SetResolution(fResolution, fResolution);

    grfx = Graphics.FromImage(bitmap);
    grfx.Clear(Color.White);
    grfx.DrawString(Text, font, Brushes.Black, 0, 0);
    grfx.Dispose();
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.DrawImage(bitmap, 0, 0);
}

The bitmap is created in the program's constructor and, as you can see, the code is rather involved. The problem is that I wanted the bitmap to be the exact size of the text string it displays, but I didn't necessarily want to use a bitmap resolution associated with a real display device. Do you see the problem? You need MeasureString to get the dimensions of a text string, and you need a Graphics object to use MeasureString. But if you want that Graphics object to be based on a bitmap with an arbitrary device resolution, you need a Bitmap object or an Image object to get that Graphics object!

For that reason, the constructor creates two bitmaps. The first one is tiny, just 1 pixel square, but that's enough. The program assigns this tiny bitmap a 300-dpi resolution by using the fResolution constant. It obtains a Graphics object, creates a 72-point Times New Roman font, and then calls MeasureString.

The MeasureString dimensions are used to create a new bitmap. The bitmap must have the same 300-dpi resolution. The program then obtains a Graphics object for this bitmap, clears the background to white, and draws the text in black.

The program uses DrawImage to display the same bitmap on both the video display and the printer. The resulting image looks like a normal 72-point font. But the real proof that something interesting is happening here is the printer output: the font characters appear as round and unjagged as any other 300-dpi output. Try changing the fResolution constant to something much smaller (say, 30 dpi), and witness the dramatic difference on both the video display and the printer.
The Shadow Bitmap

Occasionally, implementing an `OnPaint` method can be costly in terms of processing time or memory. The client area could contain a complex image that has been assembled over a long period of time, for example. For such applications, implementing a **shadow bitmap** is usually an excellent solution. A shadow bitmap is a bitmap that your program draws on whenever it also draws on its client area outside the `OnPaint` method. Then the `OnPaint` method reduces to a simple call to `DrawImage`.

In Chapter 8, "Taming the Mouse," I presented a program named Scribble that let you do free-form drawing on your client area with the mouse. At the time, I was able to show only one solution to saving the drawing for refreshing during the `OnPaint` method. That was the ScribbleWithSave program (on page 356), which saved every coordinate point in `ArrayList` objects. There's nothing wrong with this approach! In fact, if you wanted to give the user the option to edit the drawing by manipulating individual lines, saving every one of those coordinate points would be necessary. ScribbleWithSave is the first step to creating a drawing program that saves the drawing in a metafile format.

The following new version of the Scribble program is called ScribbleWithBitmap and maintains the entire image in a large bitmap. This program might be the first step in creating a paint program.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ScribbleWithBitmap: Form
{
    bool bTracking;
    Point ptLast;
    Bitmap bitmap;
    Graphics grfxBm;

    public static void Main()
    {
        Application.Run(new ScribbleWithBitmap());
    }

    public ScribbleWithBitmap()
    {
        Text = "Scribble with Bitmap";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        // Create bitmap

        Size size = SystemInformation.PrimaryMonitorMaximizedWindowSize;
        bitmap = new Bitmap(size.Width, size.Height);

        // Create Graphics object from bitmap
    }
}```
protected override void OnMouseDown(MouseEventArgs mea)
{
    if (mea.Button != MouseButtons.Left)
        return;

    ptLast = new Point(mea.X, mea.Y);
    bTracking = true;
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    if (!bTracking)
        return;

    Point ptNew = new Point(mea.X, mea.Y);

    Pen pen = new Pen(ForeColor);
    Graphics grfx = CreateGraphics();
    grfx.DrawLine(pen, ptLast, ptNew);
    grfx.Dispose();

    // Draw on bitmap

    grfxBm.DrawLine(pen, ptLast, ptNew);

    ptLast = ptNew;
}

protected override void OnMouseUp(MouseEventArgs mea)
{
    bTracking = false;
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    // Display bitmap

    grfx.DrawImage(bitmap, 0, 0, bitmap.Width, bitmap.Height);
}

The statements that I've added to the basic Scribble program are highlighted with comments: there aren't many of them. In the constructor, I create a bitmap with a size obtained from
The ScribbleWithBitmap version of the program is considerably shorter and sweeter than ScribbleWithSave. But this simplicity comes at a cost: In a very real sense, the coordinate points of the polylines have been lost. They can't easily be extracted again from the bitmap.

And here's another difference: The efficiency of the OnPaint method in ScribbleWithSave depends on how complex the drawing is. As more and more polylines are added to the total drawing, OnPaint will require longer to redraw them. The speed of the OnPaint method in ScribbleWithBitmap is independent of the complexity of the image.

As I mentioned, the ScribbleWithBitmap program creates a bitmap the size of a maximized window in its constructor. This is a fairly good approximation of how large such a shadow bitmap should be. However, if the user changes the display size while ScribbleWithBitmap is running, the bitmap could become inadequate. To deal with this eventuality, you need to install a handler for the DisplaySettingsChanged event in the SystemEvents class defined in the Microsoft.Win32 namespace. But how do you respond to the event? If the display is getting larger, you could create a new bitmap with the new size and copy the old bitmap to the new one. But what happens if the display gets smaller? Do you create a smaller bitmap and potentially throw away part of the existing image? It's not an easy problem to solve!

Binary Resources

If your application needs to display a particular bitmap, I've demonstrated numerous times in this chapter how you can load the bitmap from a file.

But as you know, storing bitmaps in separate files isn't always a good solution for an application that is distributed to other users. What happens if the program file and the bitmap file are separated? An overzealous user might be cleaning up the hard drive by erasing "unnecessary" files and suddenly, Bam! No more bitmap.

For this reason, it's often advantageous to store small binary files—particularly bitmaps, icons, and custom cursors—right in the application's .exe file. That way, they can never be lost. Files stored in the executable in this way are known as resources.

Visual C# .NET lets you create binary resources by using an interactive image editor. To add a binary resource to a program, choose the Add New Item option from the Project menu. In the Add New Item dialog box, pick Resources from the Categories list on the left. On the right, choose Bitmap File, Cursor File, or Icon File. Give the file whatever name you want.

For bitmap files, the Properties window allows you to specify the dimensions and the color format. For cursor files, the default format is 32 pixels square and 16 colors. But you can also paint in two colors known as Screen and Reverse Screen. The Screen color is transparent. You use it to make a cursor nonrectangular (as most cursors are). The Reverse Screen color reverses the color of the background behind the cursor; Reverse Screen is rarely used. You also need to specify a hot spot for cursors.

For icon files, you have 16 colors available plus Screen and Reverse Screen by default. The Screen color is often used in icons to make them nonrectangular. The Reverse Screen color was popular in the early days of Windows but is much less used nowadays.

Normally, you create an icon that is 32 pixels square and has 16 colors. But icons are often displayed in a smaller size, which is 16 pixels square. Windows will display the 32-pixel icon as a 16-pixel icon by eliminating every other row and column of pixels. If your icon doesn't look quite right after 75 percent of its content has been ripped out, you can also create a custom 16-pixel-square version that is stored in the same icon file. In the image editor, you can switch between these two formats by using the New Image Type option on the Image menu.

Now here's the most important rule whenever you create a bitmap, a cursor, or an icon file you want to use as a resource. Listen up.
In Visual C# .NET, when you select any bitmap, icon, or cursor file in Solution Explorer that is part of a project, you'll see (or you can invoke) a Properties window for the file. Change the Build Action property to Embedded Resource. That property instructs Visual C# .NET to embed the resource file in the .exe file for the project. In the program, you load such a resource by using a Bitmap constructor, a Cursor constructor, or an Icon constructor.

Let's look at an example with an icon resource. The project is called ProgramWithIcon, which means that the program file is ProgramWithIcon.cs, which means that it contains a class named ProgramWithIcon. Just to keep the rhythm going, I also named the icon file ProgramWithIcon.ico. The icon image is a simple file cabinet. The program doesn't do much except load the icon and set the form's Icon property.

ProgramWithIcon.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ProgramWithIcon: Form
{
    public static void Main()
    {
        Application.Run(new ProgramWithIcon());
    }
    public ProgramWithIcon()
    {
        Text = "Program with Icon";

        Icon = new Icon(typeof(ProgramWithIcon),
                      "ProgramWithIcon.ProgramWithIcon.ico");
    }
}
```

ProgramWithIcon.ico

Again, if you're re-creating this program yourself in Visual C# .NET, be sure to specify the Build Action for the icon file as Embedded Resource.

To load the icon, use the following constructor of the Icon class:
Other Icon constructors let you load icons from files or streams. If an icon file or resource contains multiple images, you can attempt to obtain an icon of a specific size based on an existing icon:

```
Icon(Icon icon, Size size)
Icon(Icon icon, int cx, int cy)
```

These constructors try to match the available icons with the size you specify. They won't stretch or compress icons. You use the following properties of the Icon class to determine an icon's size:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Size</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Height</td>
<td>get</td>
</tr>
</tbody>
</table>
```

You can also use these methods of the Graphics class to draw the icon on your client area:

```
void DrawIcon(Icon icon, int x, int y)
void DrawIcon(Icon icon, Rectangle rect)
void DrawIconUnstretched(Icon icon, Rectangle rect)
```

What ProgramWithIcon does, however, is simply assign the return value of the Icon constructor to the Icon property of the form. Notice the icon in the upper left corner of the form:
Now take a look at the `Icon` constructor in the program:

```csharp
new Icon(typeof(ProgramWithIcon), "ProgramWithIcon.ProgramWithIcon.ico")
```

`ProgramWithIcon` occurs so often in this project and in this statement that it's hard to tell what refers to what. Let's analyze the constructor in detail.

The first argument of the constructor refers to the `ProgramWithIcon` class. Within that `typeof` operator, you can use the name of any class that your program defines. Or you can use the name of any structure, enumeration, interface, or delegate that your program defines.

In any code in the `ProgramWithIcon` class, the expression `

typeof(ProgramWithIcon)
```
is equivalent to

```csharp
this.GetType()
```
or, more simply,

```csharp
GetType()
```

This equivalence means that you can use the somewhat shorter constructor

```csharp
new Icon(GetType(), "ProgramWithIcon.ProgramWithIcon.ico")
```

and the program still works the same.

The second argument to the `Icon` constructor is more or less a filename. If you named the icon file `MyIcon.ico`, the `Icon` constructor would look like this:

```csharp
new Icon(GetType(), "ProgramWithIcon.MyIcon.ico")
```

The first part of the quoted name is called a `namespace`, but it's a resource namespace. Don't confuse it with the .NET Framework namespace. By default, Visual C# .NET gives this resource namespace the same name as the project, but you can change it. It's the field labeled Default Namespace in the Property Pages dialog box for the project. The name in that field must agree with the first part of the quoted name in the `Icon` constructor. You can even set the Default Namespace field to nothing, in which case the second argument to the `Icon` constructor is just the bare filename

```csharp
new Icon(GetType(), "ProgramWithIcon.ico");
```
or `MyIcon.ico` or whatever you've named the file.

If you're running the C# compiler from the command line, you use the `/res` switch for each resource. For example, if you use the compiler switch

```bash
/res:ProgramWithIcon.ico
```
you load the icon like so:

```csharp
new Icon(GetType(), "ProgramWithIcon.ico")
```

Or you can give the icon an extended name following the filename and a comma:

```bash
/res:ProgramWithIcon.ico,ProgramWithIcon.ProgramWithIcon.ico
```

You then use the constructor

```csharp
new Icon(GetType(), "ProgramWithIcon.ProgramWithIcon.ico")
```
to load the icon.

Here's a problem you might run into if you just use the default resource namespace name that Visual C# .NET assigns to your project: Suppose you create a new project named ProgramWithIconPlus in which the `ProgramWithIconPlus` class inherits from the `ProgramWithIcon` class. In the `ProgramWithIconPlus` project, you create a new file named `ProgramWithIconPlus.cs` and you also add a link to the existing `ProgramWithIcon.cs` file. But you decide not to create a new icon for the new program. Instead, you create a link in the `ProgramWithIconPlus` project to the `ProgramWithIcon.ico` file. The constructor in the `ProgramWithIcon` class continues to load the icon.
And what happens when the program tries to load the icon? It terminates with an exception. So what's going on? The statement in the `ProgramWithIcon` constructor to load the icon looks like this:

```csharp
Icon = new Icon(typeof(ProgramWithIcon),
                "ProgramWithIcon.ProgramWithIcon.ico");
```

But the default resource namespace for ProgramWithIconPlus is ProgramWithIconPlus, not ProgramWithIcon. The simple solution? Change the Default Namespace field in the ProgramWithIconPlus project to ProgramWithIcon. Or make all the Default Namespace fields blank and use the naked filename in the constructor.

Loading a cursor is exactly the same as loading an icon. The `Cursor` constructor looks like this:

```csharp
Cursor Constructors (selection)
```

```csharp
Cursor(Type type, string strResource)
```

And I've already shown you this constructor for the `Bitmap` class:

```csharp
Bitmap Constructors (selection)
```

```csharp
Bitmap(Type type, string strResource)
```

And now we'll put the `Bitmap` constructor to use.

**Animation**

Windows Forms and GDI+ are missing a couple features that are usually considered important for animation. In Chapter 8, I discussed how GDI+ doesn't support exclusive-OR (XOR) drawing. XOR drawing lets you draw an image and then draw it again to erase what you drew. Another problem is that Windows Forms doesn't allow any way to read pixels from the screen. When doing animation, it's often useful to read a block of pixels from the screen as a bitmap, draw on it, and then write it back to the screen.

Still, however, you can perform some rudimentary animation in a Windows Forms program. One approach to animation is called *frame animation* and involves the successive display of identically sized bitmaps, much like a movie. Here's a program that loads in four bitmaps as resources and then uses a `Timer` event to display a winking eye.

**Wink.cs**

```csharp
//------------- ---------------------
// Wink.cs © 2001 by Charles Petzold
//-----------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Wink: Form
{
    protected Image[] aimage = new Image[4];
    protected int iImage = 0, iIncr = 1;

    public static void Main()
    {
```
Application.Run(new Wink());

public Wink()
{
    Text = "Wink";
    ResizeRedraw = true;
    BackColor = Color.White;

    for (int i = 0; i < 4; i++)
        aimage[i] = new Bitmap(GetType(),
            "Wink.Eye" + (i + 1) + ".png");

    Timer timer = new Timer();
    timer.Interval = 100;
    timer.Tick += new EventHandler(TimerOnTick);
    timer.Enabled = true;
}

protected virtual void TimerOnTick(object obj, EventArgs ea)
{
    Graphics grfx = CreateGraphics();

    grfx.DrawImage(aimage[iImage],
        (ClientSize.Width - aimage[iImage].Width) / 2,
        (ClientSize.Height - aimage[iImage].Height) / 2,
        aimage[iImage].Width, aimage[iImage].Height);

    grfx.Dispose();

    iImage += iIncr;

    if (iImage == 3)
        iIncr = -1;
    else if (iImage == 0)
        iIncr = 1;

}

Notice that the constructor loads in the four resources using the names Wink.Eye1.png, Wink.Eye2.png, and so forth. The Wink part of the name is the resource namespace. The
TimerOnTick method uses DrawImage to display each image in the center of the client area. Here’s the program caught in action:

When doing animation, you should try to display images in their pixel size to avoid stretching or compressing the image, which tends to sap the CPU.

Just for fun, I subclassed the Wink class in this program and used the RotateFlip method to make a set of right eyes out of the left eyes.

DualWink.cs

//---------------------------------------
// DualWink.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DualWink: Wink
{
    Image[] aimageRev = new Image[4];

    public new static void Main()
    {
        Application.Run(new DualWink());
    }

dualWink()
{
    Text = "Dual " + Text;

    for(int i = 0; i < 4; i++)
    {
        aimageRev[i] = (Image) aimage[i].Clone();
        aimageRev[i].RotateFlip(RotateFlipType.RotateNoneFlipX);
    }
}
protected override void TimerOnTick(object obj, EventArgs ea)
{
    Graphics grfx = CreateGraphics();

    grfx.DrawImage(aimage[iImage],
        ClientSize.Width / 2,
        (ClientSize.Height - aimage[iImage].Height) / 2,
        aimage[iImage].Width, aimage[iImage].Height);

    grfx.DrawImage(aimageRev[3 - iImage],
        ClientSize.Width / 2 - aimageRev[3 - iImage].Width,
        (ClientSize.Height - aimageRev[3 - iImage].Height) / 2,
        aimageRev[3 - iImage].Width,
        aimageRev[3 - iImage].Height);

    grfx.Dispose();

    iImage += iIncr;

    if (iImage == 3)
        iIncr = -1;

    else if (iImage == 0)
        iIncr = 1;
}
}

This project also requires links to the four PNG files associated with the Wink program. And remember my warning about inheriting from classes that load resources? In the DualWink project, I had to change the resource namespace from DualWink to Wink.
And now, what you've all been waiting for: the bouncing ball program. The Bounce program basically creates a square bitmap, draws a red ball on it, and then draws the bitmap in different places on the client area, simulating a ball that bounces off the walls.

Bounce.cs
//-------------------------------------
// Bounce.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class Bounce: Form
{
    const int iTimerInterval = 25;    // In milliseconds
    const int iBallSize = 16;         // As fraction of client area
    const int iMoveSize = 4;          // As fraction of ball size

    Bitmap bitmap;
    int xCenter, yCenter;
    int cxRadius, cyRadius, cxMove, cyMove, cxTotal, cyTotal;

    public static void Main()
    {
        Application.Run(new Bounce());
    }
    public Bounce()
    {
        Text = "Bounce";
        ResizeRedraw = true;
        BackColor = Color.White;

        Timer timer = new Timer();
        timer.Interval = iTimerInterval;
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Start();

        OnResize(EventArgs.Empty);
    }

    protected override void OnResize(EventArgs ea)
    {
        Graphics grfx = CreateGraphics();
        grfx.Clear(BackColor);

        float fRadius = Math.Min(ClientSize.Width / grfx.DpiX,
                                   ClientSize.Height / grfx.DpiY)
                        / (float)iBallSize;


cxRadius = (int) (fRadius * grfx.DpiX);
cyRadius = (int) (fRadius * grfx.DpiY);

grfx.Dispose();

cxMove = Math.Max(1, cxRadius / iMoveSize);
cyMove = Math.Max(1, cyRadius / iMoveSize);

cxTotal = 2 * (cxRadius + cxMove);
cyTotal = 2 * (cyRadius + cyMove);

bitmap = new Bitmap(cxTotal, cyTotal);

grfx = Graphics.FromImage(bitmap);
grfx.Clear(BackColor);

DrawBall(grfx, new Rectangle(cxMove, cyMove,
  2 * cxRadius, 2 * cyRadius));
grfx.Dispose();

xCenter = ClientSize.Width / 2;
yCenter = ClientSize.Height / 2;
}

protected virtual void DrawBall(Graphics grfx, Rectangle rect)
{
  grfx.FillEllipse(Brushes.Red, rect);
}

void TimerOnTick(object obj, EventArgs ea)
{
  Graphics grfx = CreateGraphics();

  grfx.DrawImage(bitmap, xCenter - cxTotal / 2,
                 yCenter - cyTotal / 2,
                 cxTotal, cyTotal);
  grfx.Dispose();

  xCenter += cxMove;
yCenter += cyMove;

  if ((xCenter + cxRadius >= ClientSize.Width) ||
      (xCenter - cxRadius <= 0))
    cxMove = -cxMove;

  if ((yCenter + cyRadius >= ClientSize.Height) ||
      (yCenter - cyRadius <= 0))
    cyMove = -cyMove;
The big question, of course, is not how the ball is drawn on the client area but how the previous image of the ball is erased—and whether the program manages to accomplish that feat without an inordinate amount of flickering. The trick here is that the bitmap is actually larger than the ball, and the extra margin around the ball is sufficient to erase the previous ball.

Bounce reconstructs the bitmap whenever the form gets a call to `OnResize`. The radius of the ball is set to 1/16 of the width or height of the client area, whichever is less. But the program constructs a bitmap that is larger than the ball. On each of its four sides, the bitmap extends beyond the ball’s dimensions by 1/4 of the radius. (You can change both these factors fairly easily.) The entire bitmap is colored white, and then the ball is drawn. (I put the ball-drawing code in a protected virtual method in hope that a future chapter will provide an override to draw a better-looking ball.)

The margins around the ball are stored as `cxMove` and `cyMove`. Not coincidentally, these two values are precisely the amount of space that the bitmap is moved on every call to `TimerOnTick`.

Such a simple approach to animation can’t work in the general case. Change the background of the client area to anything but a solid color, and the whole technique falls apart.

**The Image List**

In Chapter 12, we’ll begin working with controls, specifically buttons, labels, and scroll bars, and soon after that, menus, list boxes, edit boxes, and others. You’ll find that you can often use bitmaps on the surface of controls instead of (or in addition to) text. At the furthest extreme, a toolbar control that often appears below application menus is usually just a string of small bitmaps.

To help you deal with collections of images, the `System.Windows.Forms` namespace defines the `ImageList` class. An image list is essentially just a flexible array of `Image` objects with the same size and color format. You put images into an `ImageList` object (in a process I’ll explain shortly) and access them as if you were dealing with an array. The images that you put into the image list don’t have to be the same size when you put them in—but they will be scaled to the same size when you extract them.

These are the crucial `ImageList` properties:

<table>
<thead>
<tr>
<th>ImageList Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
</tbody>
</table>

(yCenter - cyRadius <= 0))

    cyMove = -cyMove;

)
The default `ImageSize` property value is 16 pixels square, and it doesn't get changed automatically when you start adding images to the image list. You'll probably need to set it yourself based on the size of the `Image` objects you're dealing with, and possibly also based on the resolution of the video display on which the program is running.

The `ColorDepth` property is a member of the `ColorDepth` enumeration:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth4Bit</td>
<td>4</td>
</tr>
<tr>
<td>Depth8Bit</td>
<td>8</td>
</tr>
<tr>
<td>Depth16Bit</td>
<td>16</td>
</tr>
<tr>
<td>Depth24Bit</td>
<td>24</td>
</tr>
<tr>
<td>Depth32Bit</td>
<td>32</td>
</tr>
</tbody>
</table>

The default property is `Depth8Bit`, and you'll want to manually change this property based on the images you're using. Fortunately, the `ColorDepth` enumeration is defined in a very rational manner, so if you have an `Image` object (named `image`, for example) that you want to store in the image list, you can obtain the pixel format, obtain the number of bits per pixel by using the static `Image.GetPixelFormatSize` method, and cast that to a value of type `ColorDepth`:

```csharp
imglst.ColorDepth = (ColorDepth)Image.GetPixelFormatSize(image.PixelFormat);
```

I know that fourth property named `Images` looks scary because the type of this property is defined as `ImageList.ImageCollection`. That long name only means that it's a class named `ImageCollection` that's defined in the `ImageList` class. In an application, you'll never have to refer to the `ImageCollection` class: you need only refer to the `Images` property to use properties and methods of the `ImageCollection` class. The `Images` property is what stores all the images in the image list.

The functionality of the `Images` property shows up in a number of other classes in `System.Windows.Forms`. In Chapter 12, you'll encounter a property of the `Control` class named `Controls` that is of type `Control.ControlCollection`. In Chapter 14, you'll see a property of the `Menu` class named `MenuItems` that is of type `Menu.MenuItemCollection`. All these properties work pretty much the same. The types of the properties all implement the `IList`, `ICollection`, and `IEnumerable` interfaces (defined in the `System.Collections` namespace), which allow these properties to work like expandable arrays.

To create an object of type `ImageList`, you call the default constructor:

```csharp
ImageList imglst = new ImageList();
```

You'll then want to set the `ImageSize` and `ColorDepth` properties. You add `Image` objects to the image list by using one of the following methods:

```csharp
void Add(Image image)
void Add(Image image, Color clrTransparent)
```
```csharp
void Add(Icon icon)
void AddStrip(Image image)

Because these methods are defined in the `ImageList.ImageCollection` class, you call them by using the `Images` property of the `ImageList` object. It's actually a lot simpler than it sounds:
```csharp
imglist.Images.Add(image);
```n
As you add each image, it is assigned an index beginning at 0. The `AddStrip` method adds multiple images, the number of which depends on the width of the image you pass to the method and the width of the `ImageSize` property.

The number of images in an `ImageList` is indicated by the following property:

<table>
<thead>
<tr>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int Count</code></td>
<td>get</td>
</tr>
</tbody>
</table>

Use `Count` in an expression like this:
```csharp
imglst.Images.Count
```

Most important, you can index the `Images` property as if it were an array. The expression
```csharp
imglst.Images[2]
```
returns the third `Image` object in the image list. If the image list has fewer than three images, the expression throws an exception. You can also replace an image in the image list by using indexing:
```csharp
```

However, if the image list doesn't already contain four images, the statement throws an exception.

You can also determine whether a particular image is in the image list, and you can obtain the index of such an image:

```csharp
bool Contains(Image image)
int IndexOf(Image image)
```

For example, the expression
```csharp
imglst.Images.IndexOf(image)
```
returns 2 if the object `image` is the third image in the image list, and –1 if it isn't in the image list at all. You can also remove images from the image list, either individually or entirely:

```csharp
void RemoveAt(int index)
void Clear()
```

As you'll see in later chapters, you can use an `ImageList` object in conjunction with various controls, most importantly with the `ToolBar` control. Used by itself, `ImageList` is also a convenient way to store a number of images of the same size. The `ImageList` class provides the `Draw` method to draw these images:
**ImageList Draw Methods**

```csharp
void Draw(Graphics grfx, Point pt, int index)
void Draw(Graphics grfx, int x, int y, int index)
void Draw(Graphics grfx, int x, int y, int cx, int cy, int index)
```

Notice that the index of the image in the image list is given in the last argument. For example, `imglst.Draw(grfx, x, y, 1)` draws the second image in the image list.

Watch out for the coordinates you pass to the `Draw` methods: The coordinate point passed to the first two `Draw` methods is in device units (pixels). The size of the image is based on the `ImageSize` property of the `ImageList` object, again in device units. Neither the page transform nor the world transform will affect these two methods! The use of device coordinates is intended to maximize performance but results in the postage-stamp effect on the printer. In the third `Draw` method, both the coordinate point and the width and height are in world coordinates.

**The Picture Box**

Another image-related control class is `PictureBox`. The `PictureBox` class is descended from `Control` (and hence can process keyboard and mouse input), but usually the control does little more than display an image. Here are the crucial `PictureBox` properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Image</code></td>
<td><code>Image</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>BorderStyle</code></td>
<td><code>BorderStyle</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>PictureBoxSizeMode</code></td>
<td><code>SizeMode</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

The members of the `BorderStyle` enumeration govern the border displayed around the image:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>FixedSingle</td>
<td>1</td>
</tr>
<tr>
<td>Fixed3D</td>
<td>2</td>
</tr>
</tbody>
</table>

The default is `None`. `PictureBoxSizeMode` is an enumeration that indicates how the image is displayed in the control:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>StretchImage</td>
<td>1</td>
</tr>
<tr>
<td>AutoSize</td>
<td>2</td>
</tr>
<tr>
<td>CenterImage</td>
<td>3</td>
</tr>
</tbody>
</table>

The default is `Normal`. With `PictureBox`, as with other controls, you normally use the `Location` property to set the location of the control relative to its parent and the `Size` property to set the width and height of the control. If you specify a `SizeMode` of `PictureBoxSizeMode.Normal` or
**PictureBoxSizeMode.CenterImage**, the image is displayed in its pixel size (not metrical size) within the picture box.

For **PictureBoxSizeMode.Normal**, the image is aligned with the top left of the control. If the control is larger than the pixel size of the image, you'll see the control BackColor around the right and bottom of the image. If the control is smaller than the image, part of the right and bottom of the image is hidden.

For **PictureBoxSizeMode.Centered**, the image is centered within the control. But the image is still displayed in its pixel size, so the image may be surrounded by a border, or the sides of the image may be hidden, depending on the size of the image and the size of the control.

If you set the **ClientSize** property of the **PictureBox** control equal to the **Size** property of the **Image** object, the control will be perfectly sized for the image. (The **ClientSize** property of the control indicates the size within the border.) Or you can use **PictureBoxSizeMode.AutoSize** to make the control size dependent on the **Image** size.

The **PictureBoxSizeMode.StretchImage** mode stretches the image to fit the size of the control. As you may fear, however, the image will be distorted if the aspect ratio of the control doesn't match that of the image.

So where's the **PictureBoxSizeMode** member that stretches the image isotropically? Alas, there isn't one. I'm forced to correct that deficiency with a **PictureBoxPlus** control that overrides **PictureBox** and adds a **NoDistort** property.

### PictureBoxPlus.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsWithCSharp
{
    class PictureBoxPlus: PictureBox
    {
        bool bNoDistort = false;

        public bool NoDistort
        {
            get { return bNoDistort; }
            set
            {
                bNoDistort = value;
                Invalidate();
            }
        }

        protected override void OnPaint(PaintEventArgs pea)
        {
            if ((Image != null) && NoDistort &&
                (SizeMode == PictureBoxSizeMode.StretchImage))
                ScaleImageIsotropically(pea.Graphics, Image,
```
void ScaleImageIsotropically(Graphics grfx, Image image, Rectangle rect)
{
    SizeF sizef =
        new SizeF(image.Width / image.HorizontalResolution,
            image.Height / image.VerticalResolution);
    float fScale = Math.Min(rect.Width / sizef.Width,
            rect.Height / sizef.Height);
    sizef.Width *= fScale;
    sizef.Height *= fScale;

    grfx.DrawImage(image,
            rect.X + (rect.Width - sizef.Width) / 2,
            rect.Y + (rect.Height - sizef.Height) / 2,
            sizef.Width, sizef.Height);
}

Only if SizeMode is PictureBoxSizeMode.StretchImage and the NoDistort property is true will this control display the image using the trusty ScaleImageIsotropically method. Otherwise, the control calls the OnPaint method in the base class.

Here's a program that tests the PictureBoxPlus control by using the control's Dock property to make the control fill the form's client area. The program is functionally similar to the ImageScaleIsotropic program.

PictureBoxPlusDemo.cs
//-------------------------------------------------
// PictureBoxPlusDemo.cs © 2001 by Charles Petzold
//-------------------------------------------------
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class PictureBoxPlusDemo: Form
{
    public static void Main()
    {
        Application.Run(new PictureBoxPlusDemo());
    }
    
    public PictureBoxPlusDemo()
{ 
    Text = "PictureBoxPlus Demo";

    PictureBoxPlus picbox = new PictureBoxPlus();
    picbox.Parent = this;
    picbox.Dock = DockStyle.Fill;
    picbox.Image = Image.FromFile("..\..\..\..\..\Apollo11FullColor.jpg");
    picbox.SizeMode = PictureBoxSizeMode.StretchImage;
    picbox.NoDistort = true;
}
}
Chapter 12: Buttons and Labels and Scrolls (Oh My!)

Overview

Much of the ease of use of Microsoft Windows and other graphical user interfaces results from the employment of familiar and consistent visual objects. Scroll bars, menus, push buttons, radio buttons, check boxes, text-entry fields, list boxes—these are all examples of controls. Controls are to the graphical user interface what subroutines are to programming languages. Controls let you structure and modularize your applications by off-loading low-level keyboard and mouse processing.

In the early days of Windows, controls were often referred to as child windows. With the exception of menus and scroll bars, controls appeared mostly in dialog boxes. Although it was possible to put controls on an application's main window, doing so was considered unusual and was rarely done. It wasn't until the 1991 introduction of Microsoft Visual Basic that a different Windows programming paradigm was introduced. Using Visual Basic, programmers could interactively assemble a collection of controls on the main window of an application and then write code associated with these controls. This style of programming has proved useful for developing front ends for distributed applications and is also a natural for many other "front panel" types of applications (such as the Windows CD Player).

I've already shown several examples of controls in this book. I used a Panel control in Chapter 4, a Label control in Chapter 8, and—what was once considered a relatively advanced topic in Windows programming—custom controls in Chapters 8 and 10. It's now time to begin more systematically exploring the numerous predefined controls available in the .NET Framework.

Buttons and Clicks

Perhaps the archetypal control is the push button—that ubiquitous rectangular object often labeled OK, Cancel, Open, or Save. Push buttons (often referred to as command buttons) are intended to trigger an immediate action without retaining any sort of on/off indication. You press a push button by clicking it with the mouse or—if it has the input focus—by pressing the spacebar. Even if a button doesn't have the input focus, you can sometimes trigger it by pressing Enter or Esc. I'll discuss the use of the Enter key later in this chapter and talk more about both Enter and Esc in Chapter 16.

The push button is implemented by the Button class, which is one of three classes that are descended from the abstract class ButtonBase:

We'll examine the CheckBox and RadioButton classes later in this chapter.

In a Windows Forms program, you create a push button control by creating an object of type Button. By installing an event handler for the button's Click event, you can have the button notify a form when the button has been pressed. Here's a program that creates a single button. When you click the button, the form briefly displays some text.

SimpleButton.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleButton: Form
{
    public static void Main()
    {
        Application.Run(new SimpleButton());
    }
    public SimpleButton()
    {
        Text = "Simple Button";
        Button btn = new Button();
        btn.Parent = this;
        btn.Text = "Click Me!";
        btn.Location = new Point(100, 100);
        btn.Click += new EventHandler(ButtonOnClick);
    }
    void ButtonOnClick(object obj, EventArgs ea)
    {
        Graphics grfx = CreateGraphics();
        Point ptText = Point.Empty;
        string str = "Button clicked!";

        grfx.DrawString(str, Font, new SolidBrush(ForeColor), ptText);
        System.Threading.Thread.Sleep(250);
        grfx.FillRectangle(new SolidBrush(BackColor),
                               new RectangleF(ptText, grfx.MeasureString(str, Font)));

        grfx.Dispose();
    }
}

After setting its own caption bar text, the form's constructor begins by creating an object of type Button:

Button btn = new Button();

The Button class has only a default constructor.

The next task is something I frequently forget to do, so I've tried to train myself to do it as quickly as possible after creating the control. You must assign the control a parent, and one way to do that is to set the Parent property of the control:

btn.Parent = this;
The keyword this refers to the current object, which in this case is the form—the object of type SimpleButton that was created in the Main method. The button is made a child of the form.

Parents and children seem to abound in programming. There are parent and child processes, parent and child directories, and parent and child classes, to name a few examples. You might even conclude that the parent-child relationship is the primary metaphor of operating systems and programming languages. Controls must have parents too! A control without a parent isn't visible. When you set the Parent property of a control, you are actually mandating the following relationship between the parent and the control:

- The child control appears on the surface of its parent. Furthermore, the control is clipped to the surface of its parent, meaning that no part of the control can appear outside the surface of its parent.
- The child's location is specified relative to the upper left corner of the parent's client area.
- The child initially inherits some properties of its parent, specifically the Font, ForeColor, and BackColor properties.

Back to the SimpleButton program. The next statement in the constructor assigns some text to the button's Text property:

```csharp
btn.Text = "Click Me!";
```

That is the text that will appear on the surface of the button. The next statement,

```csharp
btn.Location = new Point(100, 100);
```

indicates where the upper left corner of the button is to appear relative to the upper left corner of the parent's client area. For the SimpleButton program, I just guessed at a coordinate point that I figured would be close to the center of the client area when the program started up. I'll get more precise with coordinates later on. All the location and size properties listed in Chapter 3 (page 104) are implemented in Control. For any descendent of Control other than Form, ClientSize is usually the same as Size.

Speaking of size, you might have noticed that I haven't specified a size for the button. That's because I'm hoping that the button will be created with a default size that will be suitable for our purposes. And if it isn't, getting a button with an inappropriate size will be an important lesson for us.

Finally, the form installs an event handler for the button's Click event:

```csharp
btn.Click += new EventHandler(ButtonOnClick);
```

The button generates a Click event when it is clicked with the mouse or—if it has the input focus—when the spacebar is pressed. Because we'll be installing many event handlers in the chapters ahead, I'll be using a standard naming scheme. The name of the event handler will consist of the object type (in this case Button), perhaps another descriptive word if the class has more than one handler of a particular event of a particular object type, the word On, and the name of the event, in this case Click.

The ButtonOnClick method must be defined in accordance with the EventHandler delegate. The method has two arguments:

```csharp
void ButtonOnClick(object obj, EventArgs ea)
{
    :
}
```

In the SimpleButton program, the ButtonOnClick event simply displays a line of text in the upper left corner of the client area and then erases it 1/4 second later.

Keep in mind that despite the fact that ButtonOnClick handles Click events from the button, the event handler is still a method of the SimpleButton class. For example, when the ButtonOnClick method calls CreateGraphics, it's obtaining a Graphics object that pertains to the form, not to the button. If the ButtonOnClick method wants to access a property or method of the button, it can cast the object argument like so:

```csharp
Button btn = (Button) obj;
```
Or the Button object could be stored as a field in the SimpleButton class.

It's also important not to confuse the method I've called ButtonOnClick with the normal OnClick method. If the SimpleButton program overrides the OnClick method, that method would get click events for the form, not for the button. When you click the button, the ButtonOnClick method is called. When you click anywhere else in the client area, the OnClick method is called.

Here's how the SimpleButton program comes up:

![Simple Button](image)

You can trigger the push button (and cause the ButtonOnClick method to display the text) by using the mouse or by pressing the spacebar or the Enter key.

**Keyboard and Mouse**

I've already alluded to how the SimpleButton form and the push button respond to mouse input: When the mouse cursor is positioned over the push button, the button gets the mouse events. When the mouse cursor is otherwise positioned over the client area of the form, the form gets the mouse events. (But don't forget about mouse capturing: if you press the mouse button over the push button, the push button captures the mouse and continues to receive all mouse events until the mouse button is released or the push button loses the mouse capture.)

With regard to keyboard input, the difference between SimpleButton and most of the previous programs in this book is much more extreme: whenever the SimpleButton program is the active window, the button has the input focus. That means that the form gets no keyboard input.

You may want to verify this fact for yourself. If you were to include OnKeyDown, OnKeyUp, and OnKeyPress overrides in the SimpleControl class, they would reveal that the form itself gets no keystroke events. You could also install KeyDown, KeyUp, and KeyPress event handlers for the button. You'd need to add methods to the SimpleButton class that look like this:

```csharp
void ButtonOnKeyDown(object obj, KeyEventArgs kea)
{
    ;
}
```

You'd also need to install event handlers for the button, like so:

```csharp
btn.KeyDown += new KeyEventHandler(ButtonOnKeyDown);
```

If you were to install keyboard event handlers for the button, you'd find that when SimpleButton is the active program, most of the keystrokes go to the button. The button itself ignores most of these events except when the spacebar is pressed.
I said that the button gets most of the keystrokes. There are a few keystrokes that neither the form nor the button control see at all. The missing keyboard events are the KeyDown events for the Enter key, the Tab key, and the arrow keys (up, down, left, and right), and the KeyPress events for the Enter and Tab keys.

It's no coincidence that the Tab key and the arrow keys constitute the normal keyboard interface for navigating among controls in Windows dialog boxes and that the Enter key is normally used for triggering the default push button. These missing keystrokes are consumed in code implemented in the ContainerControl class. You might recall that ContainerControl is one of the ancestors of Form. ContainerControl is the class responsible for implementing focus management among child controls. For this reason, a class such as Form that is responsible for maintaining a collection of controls is often called a container of the controls.

A control gets keyboard and mouse input only if it is both visible and enabled, which means that both the Visible and Enabled properties are set to true:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Visible</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Enabled</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Both properties are true by default. If you set the Visible property to false, the control disappears from view and doesn't receive keyboard or mouse input. If the control is on a form, the mouse input that would have gone to the control goes to the form instead, just as if the control weren't there. If the nonvisible control is the only control on the form, keyboard input goes to the form as usual.

If you set the Enabled property of a control to false, the control is still visible but it doesn't respond to keyboard or mouse input. Often a control indicates that it's disabled by displaying text in a faint gray color.

**Control Issues**

Although SimpleButton may appear to be quite a simple program, I'm actually doing something very controversial in it. In fact, some people might look at such a program and shake their heads in exasperation. Can you believe that? What is it I'm doing that could possibly be so bad?

I'm coding.

As you may know, Visual C# .NET includes the Windows Forms Designer, which lets you interactively design your form by selecting and positioning controls, and then writing code associated with each control. The Windows Forms Designer generates code in your .cs file that creates these controls and sets their properties. The advantage of the Windows Forms Designer is obvious: you get to move the controls around to an aesthetically pleasing configuration without getting involved in the actual numbers that specify their location and size.

In this book, however, for the most part I'm going to pretend that the Windows Forms Designer doesn't exist. All my controls will be manually coded. There are some definite advantages to manually coding the locations and sizes of controls: You can use const values. You can use variables. You can use arrays. You can use for loops. In the pages ahead, I'll demonstrate these options.

More important, however, is that the Windows Forms Designer will take you only so far. At some point, you'll need to write code, and you'll probably also need to understand the code that the Windows Forms Designer is generating.

Don't misunderstand me. There's nothing wrong with using the Windows Forms Designer. But there is something wrong with not being able to code the controls yourself.

**Deeper into Buttons**

I didn't set the Size property of the Button object I created in the SimpleButton program, and nothing bad seemed to happen. On my machine, and probably on yours too, the button came out
approximately the correct size. However, if your display settings include a font much larger than
Large Fonts, it's possible that some of the button text was clipped. And that's no good.

When you decorate a form with controls, your primary design criteria must be to make it usable.
Usability involves a number of factors. The controls shouldn't be too crowded, for example. They
should be logically ordered. It helps if the form is aesthetically attractive in some way. But above all,
control text can't be clipped! A user might figure out that "Cance" is really "Cancel," but it doesn't
reflect well on you, the programmer.

You can size controls in a couple ways, which I'll be explaining in this chapter. Your experience with
creating device-independent graphics involving text will certainly help in this regard. But sizing
controls involves some intangibles as well. For example, if you set the height of a push button to the
font height, the text will be clipped anyway because of the button border. The border of a push
button is 4 pixels wide, but this information isn't available anywhere, and the size might be different
in extreme device resolutions.

So how tall should a push button be? Traditionally, a height of 7/4 (or 175 percent) of the font height
has been recommended. The next program in this chapter uses twice the font height, and those
buttons certainly don't look too large. Testing your programs with different display resolutions is
essential.

You'll have additional problems if you decide to translate your application to other languages.
Translation affects the width of text strings, and some languages tend to be a little wordier than
others.

The following program has two push buttons to demonstrate button sizing. One button makes the
form 10 percent larger; the other makes it 10 percent smaller (within limits imposed by Windows).
The buttons remain positioned in the center of the client area.

TwoButtons.cs
//--------------------------------------------------------
// TwoButtons.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoButtons: Form
{
    readonly Button btnLarger, btnSmaller;
    readonly int    cxBtn, cyBtn, dxBtn;

    public static void Main()
    {
        Application.Run(new TwoButtons());
    }
    public TwoButtons()
    {
        Text = "Two Buttons";
        ResizeRedraw = true;

        cxBtn = 5 * Font.Height;
        cyBtn = 2 * Font.Height;
        dxBtn = Font.Height;
btnLarger = new Button();
btnLarger.Parent = this;
btnLarger.Text   = "&Larger";
btnLarger.Size   = new Size(cxBtn, cyBtn);
btnLarger.Click += new EventHandler(ButtonOnClick);

btnSmaller = new Button();
btnSmaller.Parent = this;
btnSmaller.Text   = "&Smaller";
btnSmaller.Size   = new Size(cxBtn, cyBtn);
btnSmaller.Click += new EventHandler(ButtonOnClick);

OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    btnLarger.Location =
        new Point(ClientSize.Width / 2 - cxBtn - dxBtn / 2,
            (ClientSize.Height - cyBtn) / 2);
    btnSmaller.Location =
        new Point(ClientSize.Width / 2 + dxBtn / 2,
            (ClientSize.Height - cyBtn) / 2);
}
void ButtonOnClick(object obj, EventArgs ea)
{
    Button btn = (Button) obj;

    if (btn == btnLarger)
    {
        Left   -= (int)(0.05 * Width);
        Top    -= (int)(0.05 * Height);
        Width  += (int)(0.10 * Width);
        Height += (int)(0.10 * Height);
    }
    else
    {
        Left   += (int)(Width  / 22f);
        Top    += (int)(Height / 22f);
        Width  -= (int)(Width  / 11f);
        Height -= (int)(Height / 11f);
    }
}
The constructor calculates three values that it stores in fields: cxBtn and cyBtn indicate the width and height of each button, and dxBtn is the distance between the two buttons. All three values are based on the Height property of the form's Font property. Because controls inherit the Font property of their parents, the size of the font is applicable to the buttons as well. The buttons are made twice as high as the font height, and five times as wide. (I chose five times because it seemed to work right for this program; I'll discuss other approaches later in this chapter.) The constructor sets only the size of each button, not its location.

Because the location of each button depends on the size of the client area, the button location isn't set until the OnResize method, which is called for the first time from the last statement of the constructor.

The text of each button begins with an ampersand to indicate that the following letter should be underlined. The underlined letter functions as a keyboard accelerator. When the program runs, you'll have to momentarily press the Alt key to bring the underlines into view:

You can navigate between the buttons by using the Tab key or any of the four arrow keys. As you switch between the buttons, the dotted line just inside the button indicates the button that has the input focus. When a button has the input focus, all keystrokes (except the navigational keystrokes) go to that button. The button with the input focus generates a Click event when you press the spacebar.

The heavy outline indicates which button is the default button. The default button is the one that responds to pressing the Enter key. The difference between the button with the input focus and the default button may be a bit confusing. In this program, they are always the same button. As we begin working with other controls, however, the difference will become more apparent. Yes, the push button with the input focus is always the default button. However, if another type of control has the input focus, there can also be a default push button that will respond to the Enter key. In a dialog box, the button labeled OK or Open or Save is usually the default button. That button is triggered when a nonbutton control has the input focus and Enter is pressed. In addition, the button labeled Cancel is usually the cancel button, which is triggered when the Esc key is pressed. I'll explore these issues more in Chapter 16.

You can also trigger a button by pressing the underlined letter: L for Larger or S for Smaller. The button responds with a Click event, but the input focus doesn't change.

The ButtonOnClick method begins by casting the object argument to an object of type Button. The method can then determine what to do by comparing that object with the btnLarger and btnSmaller objects that the constructor saved as fields. Depending on which button was pressed, the method responds by increasing or decreasing the size of the window by 10 percent and also by moving the window 5 percent so that it stays in the same location on the screen.
Changing the size of the window generates a call to the `OnResize` method, which responds by moving the buttons to the new center of the client area. I could have set the location during the `Click` event after recalculating the client size, but that would prevent the program from relocating the buttons when the user manually resizes the window.

Is it normal for a program to reposition controls based on the size of the client area? No, it's not. But it's an option that becomes apparent only when you break out of the walls in which the Windows Forms Designer imprisons you. This is why we write code to begin with: to be flexible.

**Appearance and Alignment**

By default, buttons (and other controls) inherit their `Font`, `ForeColor`, and `BackColor` properties from their parent. If your program creates some child controls and then changes the font, foreground color, or background color of the form, the child controls will also reflect these changes. For example, in the TwoButtons program, you could insert the statement

```csharp
BackColor = Color.Blue;
```

anywhere in the form's constructor, or even in the `ButtonOnClick` method. This statement will turn the form's background blue, and the buttons will also inherit a blue background.

But you can also set the `Font, ForeColor, or BackColor` property of one of the buttons, for example,

```csharp
btnSmaller.BackColor = Color.Red;
```

Once that statement is executed, the background color of the `btnSmaller` button will be set to red. (You can set either the `ForeColor` or the `BackColor` property to `Color.Transparent` or another transparent or partially transparent color.) Now what happens if the statement

```csharp
BackColor = Color.Magenta;
```

is executed? The form's background changes to magenta and the `btnLarger` button background changes to magenta, but the `btnSmaller` button background remains red.

How does that work? The control actually keeps track of which property the program has set and won't override that property when the corresponding property of the parent changes. The following methods restore the control to its default state, that is, inheriting these properties from its parent:

```csharp
Control

void ResetFont();
void ResetForeColor();
void ResetBackColor();
```

The `TextAlign` property defined in the `ButtonBase` class determines how the text is oriented within the button:

```csharp
ButtonType Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContentAlignment</td>
<td>TextAlign</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

You set the property to one of the `ContentAlignment` enumeration values. Oddly enough, `ContentAlignment` is defined in the `System.Drawing` namespace but isn't used in conjunction with any class in that namespace.

**ContentAlignment Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopLeft</td>
<td>0x0001</td>
</tr>
<tr>
<td>TopCenter</td>
<td>0x0002</td>
</tr>
</tbody>
</table>
Although these values appear to be bit flags, that idiosyncrasy is a legacy from a prerelease version of Windows Forms. Do not combine `ContentAlignment` values! The default `TextAlign` value for push buttons is `MiddleCenter`, which doesn’t become apparent unless the button is somewhat larger than the text inside.

The `ButtonBase` class includes another property that affects the button’s appearance:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FlatStyle</code></td>
<td>FlatStyle</td>
<td>get/set</td>
</tr>
</tbody>
</table>

`FlatStyle` is an enumeration defined in the `System.Windows.Forms` namespace:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Flat</code></td>
<td>0</td>
<td>Flat rather than 3D</td>
</tr>
<tr>
<td><code>Popup</code></td>
<td>1</td>
<td>3D effect on mouse hovering</td>
</tr>
<tr>
<td><code>Standard</code></td>
<td>2</td>
<td>Normal 3D appearance</td>
</tr>
<tr>
<td><code>System</code></td>
<td>3</td>
<td>Operating system standard</td>
</tr>
</tbody>
</table>

The default is `FlatStyle.Standard`.

Here’s a program that displays push buttons in all four styles. Notice how the program uses the static `Enum.GetValues` method for obtaining an array of all the `FlatStyle` values. The program uses each enumeration value to assign both the `FlatStyle` and `Text` properties of the buttons.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ButtonStyles : Form
{
    public static void Main()
    {
        Application.Run(new ButtonStyles());
    }
```
public ButtonStyles()
{
    Text = "Button Styles";

    int y = 0;

    foreach (FlatStyle fs in Enum.GetValues(typeof(FlatStyle)))
    {
        Button btn = new Button();
        btn.Parent = this;
        btn.FlatStyle = fs;
        btn.Text = fs.ToString();
        btn.Location = new Point(50, y += 50);
    }
}

Here's what the four styles look like:

The Standard style is the same as System, but in this screen shot, the button labeled Standard has the input focus and is thus the default button.

**Buttons with Bitmaps**

Although you can set custom fonts and colors in your buttons, you may want to go to greater extremes in presenting a unique visual interface to your users. You can put a graphical image on your buttons in two ways. The first involves four properties of the `ButtonBase` class:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Image</td>
<td>get/set</td>
</tr>
<tr>
<td>ImageList</td>
<td>ImageList</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>ImageIndex</td>
<td>get/set</td>
</tr>
</tbody>
</table>
These properties let you specify a bitmap image to be displayed in the background of the button. Either you set the Image property to a specific Image or Bitmap object, or you set ImageList to an ImageList object and ImageIndex to an index within that list. The default ImageAlign property is ContentAlignment.MiddleLeft.

You can obtain these images in whatever manner is convenient—resources, files, or fabricated right in the program. You should set the width and height of the button equal to the width and height of the bitmap plus 8 (4 pixels for each border).

Although it's most common for a program to use a bitmap as an alternative to text, the two aren't mutually exclusive. If you also set the Text property, the text is displayed on top of the image.

Here's a version of the TwoButtons program that loads a couple 64-pixel-square images stored as resources for symbolizing the functionality of the buttons.

**BitmapButtons.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class BitmapButtons : Form
{
    readonly int cxBtn, cyBtn, dxBtn;
    readonly Button btnLarger, btnSmaller;

    public static void Main()
    {
        Application.Run(new BitmapButtons());
    }
    public BitmapButtons()
    {
        Text = "Bitmap Buttons";
        ResizeRedraw = true;

        dxBtn = Font.Height;

        // Create first button.
        btnLarger = new Button();
        btnLarger.Parent = this;
        btnLarger.Image = new Bitmap(GetType(), "BitmapButtons.LargerButton.bmp") ;
    }
    //---
    // BitmapButtons.cs © 2001 by Charles Petzold
    //---
    using System;
    using System.Drawing;
    using System.Windows.Forms;
}
```
// Calculate button dimensions based on image dimensions.

cxBtn = btnLarger.Image.Width + 8;
cyBtn = btnLarger.Image.Height + 8;

btnLarger.Size = new Size(cxBtn, cyBtn);
btnLarger.Click += new EventHandler(ButtonLargerOnClick);

// Create second button.

btnSmaller = new Button();
btnSmaller.Parent = this;
btnSmaller.Image = new Bitmap(GetType(),
    "BitmapButtons.SmallerButton.bmp");
btnSmaller.Size = new Size(cxBtn, cyBtn);
btnSmaller.Click += new EventHandler(ButtonSmallerOnClick);

OnResize(EventArgs.Empty);
}

protected override void OnResize(EventArgs ea) {
{ base.OnResize(ea);

    btnLarger.Location =
    new Point(ClientSize.Width / 2 - cxBtn - dxBtn /
        2,
    (ClientSize.Height - cyBtn) / 2);
    btnSmaller.Location =
    new Point(ClientSize.Width / 2 + dxBtn / 2,
        (ClientSize.Height - cyBtn) / 2);
}

void ButtonLargerOnClick(object obj, EventArgs ea) {
{ Left -= (int)(0.05 * Width);
Top -= (int)(0.05 * Height);
Width += (int)(0.10 * Width);
Height += (int)(0.10 * Height);
}

void ButtonSmallerOnClick(object obj, EventArgs ea) {
{ Left += (int)(Width / 22f);
Top += (int)(Height / 22f);
Width -= (int)(Width / 11f);
Height -= (int)(Height / 11f);
}
}
The program calculates the cxBtn and cyBtn dimensions based on the size of the bitmap image plus 8. After creating each button, the constructor loads bitmap resources and sets the Image property of the button. Here's what the buttons look like:

Multiple Handlers or One?

In the TwoButtons program, I have a single method that handles the Click events from both buttons. In the BitmapButtons program, I use two separate event handlers. As you begin developing forms and dialog boxes with many controls, you'll undoubtedly ponder which approach is best: to have one event handler for a collection of controls or to separate event handlers for each control.

Neither approach is entirely right or wrong. You'll probably write neater and more maintainable code if you have separate handlers for each control. However, if event handlers for several controls must share some code, it's probably best to consolidate those handlers into one.

Drawing Your Own Buttons

Specifying a bitmap image in a button isn't the only approach to displaying custom buttons. You can also take over button painting entirely by installing an event handler for the button's Paint event. This approach is sometimes called owner draw: your program is the owner of the button and it—rather than the buttons—does the drawing.

Owner-draw isn't quite as easy as using a bitmap, but it's probably a better approach if you're using simple vector drawing for the image, merely because vector drawing scales better.

What helps in creating owner-draw buttons is that you don't have to draw every single little thing. The System.Windows.Forms namespace contains a class named ControlPaint that includes a bunch of
static methods for drawing various pieces of common controls. For drawing push buttons, for example, the following two overloaded methods are useful:

**ControlPaint Static Methods (selection)**

- `void DrawButton(Graphics grfx, int x, int y, int cx, int cy, ButtonState bs)`
- `void DrawButton(Graphics grfx, Rectangle rect, ButtonState bs)`
- `void DrawFocusRectangle(Graphics grfx, Rectangle rect)`
- `void DrawFocusRectangle(Graphics grfx, Rectangle rect, Color clr, Color clr)`

The `DrawButton` method really just draws the border around the button. The `ButtonState` enumeration is a collection of bit flags that governs the appearance of the button:

**ButtonState Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0x0000</td>
</tr>
<tr>
<td>Inactive</td>
<td>0x0100</td>
</tr>
<tr>
<td>Pushed</td>
<td>0x0200</td>
</tr>
<tr>
<td>Checked</td>
<td>0x0400</td>
</tr>
<tr>
<td>Flat</td>
<td>0x4000</td>
</tr>
<tr>
<td>All</td>
<td>0x4700</td>
</tr>
</tbody>
</table>

I use both these methods in the OwnerDrawButtons program.

**OwnerDrawButtons.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class OwnerDrawButtons : Form
{
    readonly int cxImage, cyImage;
    readonly int cxBtn, cyBtn, dxBtn;
    readonly Button btnLarger, btnSmaller;
    
    public static void Main()
    {
        Application.Run(new OwnerDrawButtons());
    }
    public OwnerDrawButtons()
    {
```
Text = "Owner-Draw Buttons";
ResizeRedraw = true;

cxImage = 4 * Font.Height;
cyImage = 4 * Font.Height;
cxBtn = cxImage + 8;
cyBtn = cyImage + 8;
dxBtn = Font.Height;

btnLarger = new Button();
btnLarger.Parent = this;
btnLarger.Size = new Size(cxBtn, cyBtn);
btnLarger.Click += new EventHandler(ButtonLargerOnClick);
btnLarger.Paint += new PaintEventHandler(ButtonOnPaint);

btnSmaller = new Button();
btnSmaller.Parent = this;
btnSmaller.Size = new Size(cxBtn, cyBtn);
btnSmaller.Click += new EventHandler(ButtonSmallerOnClick);
btnSmaller.Paint += new PaintEventHandler(ButtonOnPaint);

OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    btnLarger.Location =
    new Point(ClientSize.Width / 2 - cxBtn - dxBtn / 2,
              (ClientSize.Height - cyBtn) / 2);
    btnSmaller.Location =
    new Point(ClientSize.Width / 2 + dxBtn / 2,
              (ClientSize.Height - cyBtn) / 2);
}

void ButtonLargerOnClick(object obj, EventArgs ea)
{
    Left = (int)(0.05 * Width);
    Top = (int)(0.05 * Height);
    Width = (int)(0.10 * Width);
    Height = (int)(0.10 * Height);
}

void ButtonSmallerOnClick(object obj, EventArgs ea)
{
    Left = (int)(Width / 22f);
    Top = (int)(Height / 22f);
void ButtonOnPaint(object obj, PaintEventArgs pea)
{
    Button btn = (Button) obj;
    Graphics grfx = pea.Graphics;

    ControlPaint.DrawButton(grfx, 0, 0, cxBtn, cyBtn,
        (btn == (Button) GetChildAtPoint(
            PointToClient(
                MousePosition))) &&

    GraphicsState grfxstate = grfx.Save();

    grfx.TranslateTransform((cxBtn - cxImage) / 2,
        (cyBtn - cyImage) / 2);
    if (btn == btnLarger)
        DrawLargerButton(grfx, cxImage, cyImage);
    else
        DrawSmallerButton(grfx, cxImage, cyImage);

    grfx.Restore(grfxstate);

    if (btn.Focused)
        ControlPaint.DrawFocusRectangle(grfx,
            new Rectangle((cxBtn - cxImage) / 2 + cxImage / 16,
                (cyBtn - cyImage) / 2 + cyImage / 16,
                    7 * cxImage / 8, 7 * cyImage / 8));
}

void DrawLargerButton(Graphics grfx, int cx, int cy)
{
    Brush brush = new SolidBrush(btnLarger.ForeColor);
    Pen pen = new Pen(btnLarger.ForeColor);

    grfx.TranslateTransform(cx / 2, cy / 2);

    for (int i = 0; i < 4; i++)
    {
        grfx.DrawLine(pen, 0, 0, cx / 4, 0);
        grfx.FillPolygon(brush, new Point[] {
            new Point(cx / 4, -cy / 8),
            new Point(cx / 2, 0),
            new Point(cx / 4, cy / 8)});
void DrawSmallerButton(Graphics grfx, int cx, int cy)
{
    Brush brush = new SolidBrush(btnSmaller.ForeColor);
    Pen pen = new Pen(btnSmaller.ForeColor);

    grfx.TranslateTransform(cx / 2, cy / 2);

    for (int i = 0; i < 4; i++)
    {
        grfx.DrawLine(pen, 3 * cx / 8, 0, cx / 2, 0);
        grfx.FillPolygon(brush, new Point[] {
            new Point(3 * cx / 8, -cy / 8),
            new Point(    cx / 8,       0),
            new Point(3 * cx / 8,  cy / 8)});
        grfx.RotateTransform(90);
    }
    grfx.RotateTransform(90);
}

For both buttons, I install a Paint event handler named ButtonOnPaint. The event handler begins by calling DrawButton. That long expression in the middle of the method call determines whether the flag passed as the last argument to the method should be ButtonState.Normal or ButtonState.Pushed. If you examine a normal push button, you'll find that the button switches to a pushed appearance when you press the mouse button over the button. The push button retains that appearance until you release the mouse button or you move the mouse cursor away from the button. If you move the mouse cursor back over the push button with the mouse button still pressed, the push button changes back to the pressed state. The logic I chose checks that the button’s Capture property is true and that the mouse cursor position is over the control. It works!

That takes care of the border around the button. The program next makes use of the DrawLargerButton and DrawSmallerButton methods to draw the interiors. Before calling these methods, however, the program calls TranslateTransform to shift the graphics origin to the area inside the border. I bracket the graphics transform calls with calls to the Graphics object’s Save and Restore methods so that they won’t interfere with the call to DrawFocusRectangle.

The DrawLargerButton and DrawSmallerButton methods draw the interiors. Notice that these methods begin by setting the origin to the center of the image and then draw the same image four times, each time rotating 90 degrees. (I actually used these same two methods to create the bitmaps I used in the BitmapButtons program.)

The ButtonOnPaint processing concludes with a call to DrawFocusRectangle if the button has the keyboard input focus. This method draws the standard rectangular dotted line.
Dropping Anchor

All the variations of the TwoButtons program shown so far have moved the two buttons to the center of the client area whenever the client area is resized. There are times when it would surely be convenient for controls to be dynamically positioned or resized depending on the size of the client area, but you’d prefer not to handle the OnResize code yourself. You’re in luck, for Windows Forms supports two control properties that dynamically move (and even resize) controls. These properties are called Anchor and Dock:

<table>
<thead>
<tr>
<th>Control Properties (selection)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Property</td>
<td>Accessibility</td>
</tr>
<tr>
<td>AnchorStyles</td>
<td>Anchor</td>
<td>get/set</td>
</tr>
<tr>
<td>DockStyle</td>
<td>Dock</td>
<td>get/set</td>
</tr>
</tbody>
</table>

It’s very easy to confuse these two properties! They are similar in some respects, and the AnchorStyles and DockStyle enumerations are just about identical. But the effects of the two properties are quite different. (Except when they’re the same.)

Here’s the AnchorStyles enumeration:

<table>
<thead>
<tr>
<th>AnchorStyles Enumeration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>Value</td>
</tr>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>Top</td>
<td>1</td>
</tr>
<tr>
<td>Bottom</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>4</td>
</tr>
<tr>
<td>Right</td>
<td>8</td>
</tr>
</tbody>
</table>

Notice that the values are single bits. You can combine the AnchorStyles enumeration values with the C# bitwise OR operator.

You set the Anchor style for a control, not for a form. When you resize the form, the Anchor property determines which side or sides of the form the control remains the same distance from.

The default Anchor property is not AnchorStyle.None! The default is the value 5, which can be expressed as

AnchorStyles.Left | AnchorStyles.Top
The default Anchor property means that when you resize a form, the controls remain the same distance from the left and top sides of the form, which is, of course, the behavior we normally expect.

Let's see if we can rewrite the TwoButtons program to take advantage of the Anchor property.

**TwoButtonsAnchor.cs**

```csharp
// TwoButtonsAnchor.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoButtonsAnchor : Form
{
    public static void Main()
    {
        Application.Run(new TwoButtonsAnchor());
    }
    public TwoButtonsAnchor()
    {
        Text = "Two Buttons with Anchor";
        ResizeRedraw = true;

        int cxBtn = 5 * Font.Height;
        int cyBtn = 2 * Font.Height;
        int dxBtn = Font.Height;

        Button btn = new Button();
        btn.Parent = this;
        btn.Text = "&Larger";
        btn.Location = new Point(dxBtn, dxBtn);
        btn.Size = new Size(cxBtn, cyBtn);
        btn.Click += new EventHandler(ButtonLargerOnClick);

        btn = new Button();
        btn.Parent = this;
        btn.Text = "&Smaller";
        btn.Location = new Point(ClientSize.Width - cxBtn - dxBtn,
                                  ClientSize.Height - cyBtn - dxBtn);
        btn.Size = new Size(cxBtn, cyBtn);
        btn.Click += new EventHandler(ButtonSmallerOnClick);
    }

    void ButtonLargerOnClick(object obj, EventArgs ea)
    {
        Left -= (int)(0.05 * Width);
    }

    void ButtonSmallerOnClick(object obj, EventArgs ea)
    {
        Left += (int)(0.05 * Width);
    }
}
```
Top -= (int)(0.05 * Height);
Width += (int)(0.10 * Width);
Height += (int)(0.10 * Height);
}

void ButtonSmallerOnClick(object obj, EventArgs ea)
{
    Left += (int)(Width / 22f);
    Top += (int)(Height / 22f);
    Width -= (int)(Width / 11f);
    Height -= (int)(Height / 11f);
}

The biggest change is that the OnResize method is gone. But if you ever need an OnResize method in a program that makes use of the Anchor property, be sure to call base.OnResize(ea);

or else anchoring won't work. By eliminating the OnResize method, I was able to make the cxBtn, cyBtn, and dxBtn variables local to the constructor. Also, because I use two different Click event handlers for the two buttons, the button objects don't have to be stored as fields either. I use the same btn variable for creating both buttons.

The Larger button is positioned in the upper left corner of the form, and the Smaller button is in the lower right corner of the form:

![Two Buttons with Anchor](image)

The buttons aren't flush against the edges. I use the dxBtn variable (equal to the font height) to specify the distance between the buttons and the side of the client area. The Larger button retains the default Anchor property, but the Smaller button is assigned a nondefault property:


This means that the Smaller button will remain dxBtn pixels from the right and bottom sides of the client area regardless of any changes in the size of the client area. As you make the client area very small, the buttons will overlap.

I encourage you to experiment with the anchor styles. Here's what you'll find.

If the Anchor property contains AnchorStyles values indicating a pair of opposite sides, the control changes size when the form is resized. For example, if the Anchor property is AnchorStyle.Top |
AnchorStyle.Bottom, the width of the control stays the same but the height changes as you change the height of the form. That's because the same distance is maintained from the top and bottom of the control to the top and bottom of the form. If you make the form too small, it's possible for the control to be resized into nothingness.

If you set the Anchor property to a combination of AnchorStyles values for all four sides, both the width and height of the control change size as you change the size of the form.

If the Anchor property is set to just one AnchorStyle value indicating a side, for instance, AnchorStyle.Right, the distance between the control and the right side of the client area remains the same. As you change the height of the form, however, the control retains its approximate vertical position relative to the client area.

If you set the Anchor property to AnchorStyle.None, the control retains its approximate position in the client area relative to the size of the client area. For example, if you position a control in the center of the client area and set the Anchor property to AnchorStyle.None, as you resize the client area the control will remain the same size but will stay in the approximate center of the client area.

Dock Around the Clock

Now for docking. Here are the DockStyle enumeration values:

<table>
<thead>
<tr>
<th>DockStyle Enumeration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>Top</td>
<td>1</td>
</tr>
<tr>
<td>Bottom</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>3</td>
</tr>
<tr>
<td>Right</td>
<td>4</td>
</tr>
<tr>
<td>Fill</td>
<td>5</td>
</tr>
</tbody>
</table>

The first thing you should notice is that these are not bit flags. You can't combine two or more styles. The default is DockStyle.None.

When you set the Dock property to one of the four DockStyle values indicating a side, the control is positioned flush against the entire length of that side and will touch the two adjacent sides. For example, if you specify DockStyle.Top, the control will be positioned against the top of the client area and extend the full width of the client area. As you make the form wider, the control becomes wider as well.

In contrast, the process of anchoring usually doesn't cause the control to be resized. The control is resized only if the control is anchored to opposite edges and the form is resized. However, if you position a control on a particular side and make it extend the full width or height of that side, and then set the Anchor property with AnchorStyles values that combine those three sides, the effect is pretty similar to docking the control on that side.

Let's look at another version of the TwoButtons program that docks the two buttons on the top and bottom sides.

TwoButtonsDock.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;
```
class TwoButtonsDock: Form
{
    public static void Main()
    {
        Application.Run(new TwoButtonsDock());
    }

    public TwoButtonsDock()
    {
        Text = "Two Buttons with Dock";
        ResizeRedraw = true;

        Button btn = new Button();
        btn.Parent = this;
        btn.Text = "&Larger";
        btn.Height = 2 * Font.Height;
        btn.Dock = DockStyle.Top;
        btn.Click += new EventHandler(ButtonLargerOnClick);

        btn = new Button();
        btn.Parent = this;
        btn.Text = "&Smaller";
        btn.Height = 2 * Font.Height;
        btn.Dock = DockStyle.Bottom;
        btn.Click += new EventHandler(ButtonSmallerOnClick);
    }

    void ButtonLargerOnClick(object obj, EventArgs ea)
    {
        Left   -= (int)(0.05 * Width);
        Top    -= (int)(0.05 * Height);
        Width  += (int)(0.10 * Width);
        Height += (int)(0.10 * Height);
    }

    void ButtonSmallerOnClick(object obj, EventArgs ea)
    {
        Left   += (int)(Width / 22f);
        Top    += (int)(Height / 22f);
        Width  -= (int)(Width / 11f);
        Height -= (int)(Height / 11f);
    }
}

This program dispenses with $cxBtn$, $cyBtn$, and $dxBtn$ entirely and just sets the Height property of each button equal to twice the height of the default font. The Dock property positions the button against the specified side and makes it the width of the side.
Docking isn't often used with buttons! Docking makes much more sense with toolbars (docked at the top of the client area) and status bars (docked on the bottom of the client area), as I'll be demonstrating in Chapter 20. Docking also makes much more sense in a program visually structured like Windows Explorer, with a tree view control docked at the left of the client area, a list view control docked at the right of the client area, and a splitter control in between, as I'll demonstrate in Chapter 22.

What happens when you dock two controls on the same side? You'll probably be pleased to know that docking controls on the same side doesn't cause the controls to overlap one another. The controls are stacked on the edge. For example, if you use a Dock property of DockStyle.Top with both buttons in the TwoButtonsDock program, the buttons look like this:

Just offhand, it appears as if the most recently created control takes priority for actually touching the edge. In reality, the positioning is based on the z-order, a concept I'll explain shortly.

Then there's DockStyle.Fill. I used DockStyle.Fill in the AnalogClock program in Chapter 10 to make the clock control fill up the form's entire client area. Only one control should have its Dock property set to DockStyle.Fill. The control fills up the client area but won't overlap any other controls that have nondefault Dock properties set.

Keep in mind that nothing magical is happening with the Anchor and Dock properties that you couldn't do yourself during the OnResize method.
Children of the Form

The Control class includes an important and very handy read-only property named Controls:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control.ControlCollection</td>
<td>Controls</td>
<td>get</td>
</tr>
</tbody>
</table>

Although the Controls property is defined in the Control class, it is useful only in classes descended from Control that are parent to other controls, such as Form. The Control.ControlCollection type defined for this property might look a little peculiar, but it's only a public class named ControlCollection that's defined inside the Control class. In your programs, you don't need to refer to the name of this class. You only need to refer to the Controls property.

You've already seen something like this toward the end of Chapter 11. The ImageList class has a property named Images that is of type ImageList.ImageCollection. And in Chapter 14, you'll see that the Menu class has a property named MenuItems that is of type Menu.MenuItemCollection.

The Control.ControlCollection class implements the IList, ICollection, and IEnumerable interfaces (defined in the System.Collections namespace). The total effect is to make the Controls property appear to be a flexible array (similar to the ArrayList class I discussed toward the end of Chapter 8) to which you can add and delete members. The Controls property is essentially an array of all the controls that are children of the form.

Let's assume your form creates three buttons, named btn1, btn2, and btn3. You make these three buttons children of your form in the usual way:

```
btn1.Parent = this;
btn2.Parent = this;
btn3.Parent = this;
```

After these three statements are executed, you can obtain the number of controls that are children of the form by using the Count property of the form's Controls property. If those three buttons are the only children of the form, the expression

```
Controls.Count
```

returns 3. You can also index the Controls property as if it were an array. An object of type Control is returned. For example, the statement

```
Control ctrl = Controls[1];
```

sets the ctrl variable equal to the btn2 object. If you know that the element is a push button, you can cast the return value to an object of type Button:

```
Button btn = (Button) Controls[1];
```

That indexer is read-only. You can't do something like this:

```
Controls[1] = new Button(); // Won't work!
```

Child controls get into the Control.ControlCollection class automatically when they are made children of the form. But you can also put controls into the collection by using one of the following two methods:

<table>
<thead>
<tr>
<th>Control.ControlCollection Methods (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Add(Control ctrl)</td>
</tr>
<tr>
<td>void AddRange(Control[] actrl)</td>
</tr>
</tbody>
</table>

Just offhand, the statement
Controls.Add(btn1);

looks like we’re calling a static method named Add in the Controls class. Not so! Controls is a property of Form inherited from Control. The type of the property is Control.ControlCollection, a class that defines a method named Add. Calling that Add method is equivalent to

btn1.Parent = this;

The statement

Controls.AddRange(new Control[] { btn1, btn2, btn3 });

is equivalent to the three statements earlier where I assigned the Parent property of the buttons. Of course, the AddRange statement would be a lot shorter if the three buttons were an array to begin with.

You can also remove controls from the collection:

**Control.ControlCollection Methods (selection)**

void Remove(Control ctrl)
void RemoveAt(int iIndex)
void Clear()

Removing a control from the collection doesn't destroy the control. Removing a control is merely the equivalent of setting the Parent property of the control to null. The Clear method removes all the controls from the collection.

When you remove a control from the collection, the remaining controls are reindexed to close up the indices. There won't be any skipped indices; the indices always range from 0 to 1 less than the Count property.

You can also obtain the index of a particular control:

**Control.ControlCollection Methods (selection)**

bool Contains(Control ctrl)
int GetChildIndex(Control ctrl)
int GetChildIndex(Control ctrl, bool bThrowException)

You may want to use the Contains method first to check whether the collection contains the control before calling GetChildIndex. If the collection doesn't contain the control, the first version of GetChildIndex will throw an exception. The second version won't throw an exception if bThrowException is set to false. Instead, if the control isn't part of the collection, the method returns −1.

You can also assign a control a new index:

**Control.ControlCollection Methods (selection)**

void SetChildIndex(Control ctrl, int iNewIndex)

Again, the other controls are reindexed, so the indices are still consecutive, ranging from 0 to 1 less than the number of controls. If you want to give a particular control the highest index, you can specify iNewIndex as −1.
Why would you want to change indices in the control collection? Because the control collection indices aren't simply ways of accessing the individual controls. The indexing of the control collection is also the z-order of the controls.

**Z-Order**

I've alluded before to the z-order of a group of controls that are children of the same form. The term *z-order* comes from the concept of a three-dimensional coordinate space: The x and y axes are the normal horizontal and vertical coordinates. The z axis is at right angles to the screen.

Most obviously, the z-order affects the appearance of overlapping controls that have the same parent. You can see z-order at work in the TwoButtonsAnchor program when you make the window small enough for the buttons to overlap. The button labeled Larger appears visually on top of the button labeled Smaller. The z-order also affects how controls are stacked when they are docked against the same edge of the form.

The z-order is established by the order in which you assign the *Parent* property of a control to the form or the order in which you add the control to the control collection. The z-order is established programmatically and can only be changed programmatically. The z-order does not change by the user clicking on the controls.

I often get confused about what constitutes the top and bottom of the z-order, so let me spell it out clearly here. A control at the top of the z-order has the following characteristics:
- It is the *first* control to be assigned its *Parent* property or to be added to the control collection.
- It is referenced by an index of 0 in the *Controls* property.
- It is visually on top of all other controls. It's the control that gets the mouse events when the mouse cursor is positioned over the control, regardless of other controls that might occupy the same space.
- It is the control closest to the center of the client area when multiple controls are docked against the same edge of the container.

A control at the bottom of the z-order has the following characteristics:
- It is the *last* control to be assigned its *Parent* property or to be added to the control collection.
- It is referenced by an index of (*Controls.Count* - 1) in the control collection.
- It is visually underneath all other controls.
- If multiple controls are docked against the same edge of the container, it's the control on the edge.

Aside from the *SetChildIndex* method implemented in the *Control.ControlCollection* class, a container can also change the z-ordering of its children by calling either of the following two methods:

<table>
<thead>
<tr>
<th>Control Methods (selection)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void BringToFront()</td>
<td>Elevates control to top of z-order</td>
</tr>
<tr>
<td>void SendToBack()</td>
<td>Puts control at bottom of z-order</td>
</tr>
</tbody>
</table>

For example, if a form has three child controls, *btn1*, *btn2*, and *btn3*, and *btn1* is at the top of the z-order and *btn3* is at the bottom of the z-order, then

```
btn3.BringToFront()
```

puts *btn3* at the top of the z-order and *btn2* at the bottom. That's not the same as

```
btn1.SendToBack()
```

which puts *btn2* at the top of the z-order and *btn1* at the bottom.

**The Check Box**

A second type of button is the check box. A check box consists of a small box followed by a text string. When you click the control (or press the spacebar when the control has the input focus), a check mark appears in the box. When you click the control again, the check mark disappears. Unlike the push button, the check box retains an on/off state.
These are the two crucial `CheckBox` properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td><code>Checked</code></td>
<td>get/set</td>
<td>Default is <code>false</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>AutoCheck</code></td>
<td>get/set</td>
<td>Default is <code>true</code></td>
</tr>
</tbody>
</table>

The `Checked` property indicates whether or not the control is checked. You can use this property to initialize the state of the control or to obtain the state. You'll probably want to leave the `AutoCheck` property set to `true` so that the `CheckBox` control itself will toggle the state of the control as the user clicks it with the mouse.

Whenever the `Checked` property changes, the control triggers a `CheckedChanged` event.

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CheckedChanged</code></td>
<td><code>OnCheckedChanged</code></td>
<td><code>EventHandler</code></td>
<td><code>EventArgs</code></td>
</tr>
</tbody>
</table>

The `CheckedChanged` event occurs under two conditions: when `AutoCheck` is `true` and the user clicks the `CheckBox` control, and when the program changes the `Checked` property itself, perhaps in initializing the control.

If you set `AutoCheck` to `false`, your program needs to install handlers for the control's `Click` event. The `Click` event processing will probably include the following statements to toggle the check mark:

```csharp
CheckBox chkbox = (CheckBox) obj;
chkbox.Checked ^= true;
```

Do not toggle the `Checked` property in the `CheckedChanged` event handler! Doing so will generate another `CheckedChanged` event, and another, and another...

Here's a program that creates four `CheckBox` controls that let you set the bold, italic, underline, and strikeout attributes of a font.

```csharp
class CheckBoxDemo: Form
{
    public static void Main()
    {
        Application.Run(new CheckBoxDemo());
    }
    public CheckBoxDemo()
    {
        Text = "CheckBox Demo";
        CheckBox[] achkbox = new CheckBox[4];
        int cyText = Font.Height;
```
int cxText = cyText / 2;
string[] astrText = {"Bold", "Italic", "Underline", "Strikeout"};

for (int i = 0; i < 4; i++)
{
    achkbox[i] = new CheckBox();
    achkbox[i].Text = astrText[i];
    achkbox[i].Location = new Point(2 * cxText,
                                  (4 + 3 * i) * cyText / 2);
    achkbox[i].Size = new Size(12 * cxText, cyText);
    achkbox[i].CheckedChanged +=
        new EventHandler(CheckBoxOnCheckedChanged);
}
Controls.AddRange(achkbox);
}
void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    Invalidate(false);
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    FontStyle fs = 0;
    FontStyle[] afs = {FontStyle.Bold, FontStyle.Italic,
                       FontStyle.Underline, FontStyle.Strikeout};

    for (int i = 0; i < 4; i++)
    {
        if (((CheckBox) Controls[i]).Checked)
            fs |= afs[i];

        Font font = new Font(Font, fs);
        grfx.DrawString(Text, font, new SolidBrush(ForeColor), 0, 0);
    }
}

I defined an array of CheckBox controls just so I could have the opportunity to use the AddRange method of the Controls property! The cyText variable is the height of the form's (and hence the control's) Font property. I set the cxText variable to half that size to roughly approximate the average width of lowercase characters. These variables are used to set the Location and Size of each control. Although I set the height of each control to cyText, I use 150 percent of that value to space the controls. The width of 12 times cxText is sufficient for the text and the check box. Here's what the program looks like:
The processing of the `CheckedChanged` event simply invalidates the form, which generates a call to `OnPaint`. The `OnPaint` method indexes the form's `Controls` property to obtain the `Checked` property of each of the four controls and assembles a `FontStyle` variable using that information. From there, it's a simple matter to create a new `Font` object and display some text.

Because `CheckBox` inherits from `ButtonBase`, it shares some properties with the `Button` class. The `TextAlign` property for check boxes indicates how the text is aligned within the rectangle defined by the `Size` property. The default is `ContentAlignment.MiddleLeft`, which means that the text is vertically positioned in the center of the rectangle and horizontally positioned at the left (but no farther left than the check box itself, obviously). In addition, the `CheckBox` class also has a `CheckAlign` property:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContentAlignment</td>
<td>CheckAlign</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

This property indicates the position of the check box within the rectangle. The default is also `ContentAlignment.MiddleLeft`.

If you'd like to put the check boxes at the right of the text, it makes sense to set both the `TextAlign` and `CheckAlign` properties to `ContentAlignment.MiddleRight` so that the text is right-justified as well. Another interesting variation is to make the height of the control about twice the font height and to set `CheckAlign` to `ContentAlignment.TopCenter` and `TextAlign` to `ContentAlignment.BottomCenter`. That horizontally centers the box above the horizontally centered text.

The `CheckBox` class includes another property that affects the appearance of the control:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appearance</td>
<td>Appearance</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

The `Appearance` enumeration is defined like so:

```
<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>Button</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The `Button` option causes the `CheckBox` control to look like a push button, but one that retains a checked state. You'll need to increase the height of the control to accommodate the button border.
The Three-State Alternative

Most of the time, a CheckBox object is an on/off, yes/no, 1/0 type control that George Boole would have approved of. But sometimes 1 bit isn't quite enough and 2 bits are way too many. For such cases, you can put the check box into a third state.

When might you need this option? Suppose you were writing a word processor, and somewhere (perhaps in a dialog box for font selection), you have a CheckBox control labeled Italic. If the user selects some text that isn't italic, the program should initialize the CheckBox control to its unchecked state. If the text is italic, the program initializes the CheckBox to its checked state. And if the text is partially italic and partially not italic? That's a good candidate for the third state. The check is drawn in the box, but it's a light gray color.

You shouldn't confuse this third state with a disabled CheckBox control. You would disable the CheckBox control if the selected text was displayed in a font that wasn't capable of the italic style.

To use a three-state CheckBox, you need to use the following two properties:

<table>
<thead>
<tr>
<th>CheckBox Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>CheckBox</td>
</tr>
</tbody>
</table>

First, you set the ThreeState property to true. Then, instead of using the Checked property to initialize the control and to determine its current state, you use the CheckState property. The CheckState enumeration has three values:

<table>
<thead>
<tr>
<th>CheckState Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Unchecked</td>
</tr>
<tr>
<td>Checked</td>
</tr>
<tr>
<td>Indeterminate</td>
</tr>
</tbody>
</table>

If your program needs to be informed when the CheckState changes, don't install a handler for the CheckedChanged event. Install a handler for the CheckStateChanged event:

<table>
<thead>
<tr>
<th>CheckBox Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
</tr>
<tr>
<td>CheckStateChanged</td>
</tr>
</tbody>
</table>

As the user clicks on a three-state CheckBox, the control cycles through the three states.

The Label Control

Sometimes programmers wonder, Can I mix controls and graphics output on the same form? Yes, you can, and the CheckBoxDemo program proves it. Notice in CheckBoxDemo that the CheckBoxOnCheckedChanged method contains the single statement

```csharp
Invalidate(false);
```

The false argument indicates that the method won't invalidate any part of the form occupied by child controls. Using the false argument to Invalidate prevents the controls from being unnecessarily redrawn.

Although you can mix controls and graphics on the same form, it's more common for programs to use additional controls to display text and other graphics. You saw back in Chapter 4 how you can display text on a Panel control. A control specifically designed for the display of text is the Label control. The Label control has a fairly light ancestry:
Here's another version of the CheckBoxDemo program that creates a Label control for displaying the sample line of text. Rather than invalidating the form in the CheckBoxOnCheckedChanged method and using the information to display the text in the OnPaint method, this version creates a new font during CheckBoxOnCheckedChanged and just sets the Font property of the Label control.

**CheckBoxWithLabel.cs**

```csharp
//------------------------------------------------
// CheckBoxWithLabel.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckBoxWithLabel: Form
{
    Label label;

    public static void Main()
    {
        Application.Run(new CheckBoxWithLabel());
    }
    public CheckBoxWithLabel()
    {
        Text = "CheckBox Demo with Label";

        int cyText = Font.Height;
        int cxText = cyText / 2;
        string[] astrText = {"Bold", "Italic", "Underline", "Strikeout"};

        label = new Label();
        label.Parent = this;
        label.Text = Text + ": Sample Text";
        label.AutoSize = true;
        for (int i = 0; i < 4; i++)
        {
            CheckBox chkbox = new CheckBox();
            chkbox.Parent = this;
            chkbox.Text = astrText[i];
```
chkbox.Location = new Point(2 * cxText,
    (4 + 3 * i) * cyText / 2);
chkbox.Size = new Size(12 * cxText, cyText);
chkbox.CheckedChanged +=
    new EventHandler(CheckBoxOnCheckedChanged);
}
}
void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    FontStyle   fs   = 0;
    FontStyle[] afs  = { FontStyle.Bold,      FontStyle.Italic,
                         FontStyle.Underline, FontStyle.Strikeout };

    for (int i = 0; i < 4; i++)
        if (((CheckBox) Controls[i + 1]).Checked)
            fs |= afs[i];

    label.Font = new Font(label.Font, fs);
}
}

A Label control will wrap text into multiple lines if the text is longer than the width of the control. No scroll bars are provided, however. If you prefer that a Label control display only a single line of text, you have a few properties that help facilitate that:

<table>
<thead>
<tr>
<th>Label Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

The PreferredWidth and PreferredHeight properties are consistent with the information returned from MeasureString rounded to the next highest integer. Use the AutoSize property (which by default is false) to make the label's size the same as the PreferredWidth and PreferredHeight values.

The Label control supports the same four properties as the ButtonBase control for displaying bitmaps: Image, ImageList, ImageIndex, and ImageAlign. The AutoSize property does not adjust the size of the control based on the size of the image.

Two additional properties affect the appearance of Label controls:

<table>
<thead>
<tr>
<th>Label Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>BorderStyle</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

The BorderStyle property causes a border to be drawn around the label. Set the property to one of the following enumeration values:

<table>
<thead>
<tr>
<th>BorderStyle Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
</tbody>
</table>

**BorderStyle Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>FixedSingle</td>
<td>1</td>
</tr>
<tr>
<td>Fixed3D</td>
<td>2</td>
</tr>
</tbody>
</table>

The default is `BorderStyle.None`.

The `UseMnemonic` property (which by default is `true`) causes the `Label` control to suppress ampersands and underline the letter following the ampersand. But this raises a question: If a `Label` is used just to display some text or an image, why does it need a mnemonic? The purpose—as you'll find out later in this chapter—is to navigate to controls such as scroll bars, track bars, and text boxes that have no fixed text.

**Tab Stops and Tab Order**

As you've discovered, you can navigate among child controls by using the Tab key or the arrow keys. However, if you use Tab or the arrow keys with `CheckBoxWithLabel`, you'll find that the navigation skips the `Label` control. This makes sense: the `Label` control isn't intended to get input from the keyboard, so there's no reason why it should get keyboard focus.

Whether you can navigate to a control with the Tab key is governed by the `TabStop` property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td><code>TabStop</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>TabIndex</code></td>
<td>get/set</td>
</tr>
</tbody>
</table>

Buttons have a `TabStop` property of `true`; labels have a `TabStop` property of `false`.

If `TabStop` is `true`, the `TabIndex` property determines the order in which the Tab key causes focus to shift from control to control. `TabIndex` is set when you assign a parent to the control, so the `TabIndex` values initially are the same as the z-order indices. If you change the z-order, however, the `TabIndex` property doesn't change. Your program can also change the `TabIndex` independently of the z-order.

If two controls have the same `TabIndex`, the control with the lowest z-order gets the focus first.

**Identifying the Controls**

In the `CheckBoxWithLabel` program, I defined two arrays: one with the text strings for the four check boxes and the second containing the corresponding `FontStyle` enumeration values. Unfortunately, these arrays were defined in two different areas of the program. And if you change the order of the elements in one array without changing the other, the program will no longer work right.

Moreover, take a look at how the `Controls` property is indexed in the `OnPaint` method in the `CheckBoxDemo` program as compared to the `CheckBoxOnCheckedChanged` method in the `CheckBoxWithLabel` program. In the first program, the indices are 0 through 3. In the second program, however, the `Label` control is the first control made a child of the form, so it has an index of 0. The `CheckBox` controls are indices 1 through 4.

If I were to change the constructor in `CheckBoxWithLabel` so that the `Label` control is made a child of the form after the `CheckBox` controls, the program wouldn't work right. I don't have to tell you that making the `Controls` array indexing dependent on the ordering of control creation is a bad programming practice! For a few controls, it might not be so awful. But for many controls, it could easily turn into a nightmare.

There are several ways for your program to keep track of all the controls it creates. You can always save the control objects as fields, such as the `TwoButtons` program did. Or you could install different event handlers for each control. Another approach is to use a property (or something else) of the control to uniquely identify it. The `Text` property, of course, usually identifies the control, and in an
event handler, you can even use the control text as a *switch* variable to test which control is generating the event. However, if you ever wanted to change the control text, you’d have to change both the code that assigned the control *Text* property and the *switch* and *case* construction in the event handler.

If the *Text* property isn’t quite what you want for identifying the controls, what *would* you prefer? The *Control* class includes the following property that you can set to any object that’s convenient:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Tag</td>
<td>get/set</td>
</tr>
</tbody>
</table>

This property is specifically intended to identify controls. When creating a control, you can assign the *Tag* property to any object. For example, here’s a partial definition of a *CheckBox* control that’s intended to select a color:

```csharp
chkbox.Text = "Magenta";
chkbox.Tag = Color.Magenta;
```

In the event handler, you first obtain the *CheckBox* control sending the event

```csharp
CheckBox chkbox = (CheckBox) obj;
```

and then cast the *Tag* property to a *Color* object:

```csharp
Color clr = (Color) Tag;
```

If the *Tag* property is not a *Color* object, an exception will be raised. I’ll have an example of the *Tag* property in the AutoScaleDemo program coming up shortly.

What’s also nice about object-oriented programming in general (and Windows Forms in particular) is that you can easily add anything you want to the control to identify it. It’s simply a matter of inheritance.

The following program creates a new class based on *CheckBox* that is specifically intended for displaying font styles. This new class adds just one field to the *CheckBox* class: a field named *fontstyle* of type *FontStyle*. As you can see, the definition of this new class (down at the bottom of the listing) requires a minimum amount of code.

```csharp
//*****************************************************************************/
// CustomCheckBox.cs © 2001 by Charles Petzold
//*****************************************************************************/
using System;
using System.Drawing;
using System.Windows.Forms;

class CustomCheckBox: Form
{
    public static void Main()
    {
        Application.Run(new CustomCheckBox());
    }
    public CustomCheckBox()
    {
        Text = "Custom CheckBox Demo";
    }
}
```
int cyText = Font.Height;
int cxText = cyText / 2;
FontStyle[] afs = { FontStyle.Bold, FontStyle.Italic,
                   FontStyle.Underline, FontStyle.Strikeout };

Label label = new Label();
label.Parent = this;
label.Text = Text + ": Sample Text";
label.AutoSize = true;

for (int i = 0; i < 4; i++)
{
    FontStyleCheckBox chkbox = new FontStyleCheckBox();
    chkbox.Parent = this;
    chkbox.Text = afs[i].ToString();
    chkbox.fontstyle = afs[i];
    chkbox.Location = new Point(2 * cxText,
                                (4 + 3 * i) * cyText / 2);
    chkbox.Size = new Size(12 * cxText, cyText);
    chkbox.CheckedChanged +=
        new EventHandler(CheckBoxOnCheckedChanged);
}

void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    FontStyle fs = 0;
    Label label = null;

    for (int i = 0; i < Controls.Count; i++)
    {
        Control ctrl = Controls[i];

        if (ctrl.GetType() == typeof(Label))
            label = (Label) ctrl;

        else if (ctrl.GetType() == typeof(FontStyleCheckBox))
            if (((FontStyleCheckBox) ctrl).Checked)
                fs |= ((FontStyleCheckBox) ctrl).fontstyle;
    }
    label.Font = new Font(label.Font, fs);
}

class FontStyleCheckBox: CheckBox
{
    public FontStyle fontstyle;
This program now defines the array of FontStyle values in the constructor. As each FontStyleCheckBox object is created, the program assigns the FontStyle value to the fontStyle field of the object. The program also dispenses with the string array. Instead, it converts the FontStyle value to a string for the Text property. And even if an array of strings were needed, at least the two arrays would be defined side by side and could be changed (if necessary) at the same time.

Now take a look at the CheckBoxOnCheckedChanged method. The method loops through all the controls in the Controls array and determines what type of control each one is. You can use the GetType method with any object to obtain its type, and you can use the C# typeof operator with any class name to obtain its type. If the control is a FontStyleCheckBox, the program casts the control to an object of that type, and if the Checked property is true, accesses its fontStyle field. If the control is a Label, the program saves the Label object and concludes event handling by setting its Font property to the new font. This is code that can withstand changes in the array of FontStyle values and any changes in the order in which controls are created and made children of the form.

The Auto-Scale Option

The programs shown in this chapter have used the Font.Height property of the form to scale the sizes of controls that display text. For horizontal sizing, I've used one-half the font height as a generous approximation of the average character width of lowercase letters. (It's even generous for the Courier font.) Because controls inherit their parent's font, this is a perfectly acceptable method of scaling controls. If you ever want to set a different font for the form (and hence its controls), do so early in the form's constructor before obtaining the font height.

The .NET Framework offers an alternative method of scaling controls that is referred to as auto-scale. In support of auto-scale are the following two properties of the Form class:

<table>
<thead>
<tr>
<th>Form Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>Size</td>
</tr>
</tbody>
</table>

The only static method of Form is also connected with the auto-scale feature:

**Form Static Method**

SizeF GetAutoSizeSize(Font font)

The AutoScaleBaseSize property and the GetAutoSizeSize method are useful in themselves even if you don't use the auto-scale feature. They are the only source in Windows Forms of the average character width associated with a particular font. By default, AutoScaleBaseSize returns the width and height of the form's Font property; GetAutoSizeSize returns the width and height of any Font object. The width is an average based on lowercase letters of the Latin alphabet. The height is the same as the Height property of the Font object.

If you have Small Fonts specified in your display settings, AutoScaleBaseSize reports that the 8-point default Windows Forms font has a width of 5 pixels and a height of 13 pixels. For Large Fonts, AutoScaleBaseSize reports a width of 6 pixels and a height of 15. So if you want to be more accurate about positioning controls, use

cxText = AutoScaleBaseSize.Width;
cyText = AutoScaleBaseSize.Height;

I just said that AutoScaleBaseSize returns the width and height of the form's Font property. That's true. And if you set a new Font property for the form, AutoScaleBaseSize reports the width and height of the new font. However, it's true only if the form doesn't set AutoScaleBaseSize itself. If the form sets AutoScaleBaseSize (by which I mean that you the programmer write code that sets AutoScaleBaseSize), the property returns whatever the form set it to.
And here's the secret of auto-scale: If the form sets `AutoScaleBaseSize` itself, the width and height of all the form's child controls are scaled based on the ratio of the height and width of the form's `Font` property to the height and width of the `AutoScaleBaseSize` property. If the form doesn't set `AutoScaleBaseSize`, these two ratios are simply equal to 1 and no scaling takes place.

Basically, the form can use whatever coordinate system and sizes it wants when setting the `Location` and `Size` properties of its child controls. The form then uses `AutoScaleBaseSize` to indicate that coordinate system. All the scaling takes place after the form's constructor has completed based on the `Font` and `AutoScaleBaseSize` properties at that time.

The auto-scale process is confusing at first, so let's take a look at some examples.

**How the Windows Forms Designer Uses Auto-Scale**

I mentioned that you'll someday have to look at the code that the Visual C# .NET Windows Forms Designer generates, and you may need to understand certain aspects of it. Well, auto-scale is a primary example.

Let me assume that your display settings are set to Small Fonts. When you use the Windows Forms Designer to design a form, it generates code that contains the pixel dimensions you've used. For example, a `Button` control might have its `Size` property set like so:

```csharp
this.button1.Size = new System.Drawing.Size(104, 26);
```

Yes, the Windows Forms Designer is a little verbose. But that's not the point. Normally, hard-coding pixel positions and sizes of controls is just begging for trouble. You're practically guaranteeing that somebody is going to run this program with Large Fonts or even larger fonts, and the text in the button will be truncated. But the Windows Forms Designer also adds the following statement to the constructor:

```csharp
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
```

That size is the font width and height associated with Small Fonts. What the Windows Forms Designer is essentially doing here is embedding into the code the underlying display resolution upon which all the control locations and sizes are based.

If such a program runs on another machine that has a display setting of Small Fonts, the ratios of the font width and height to the `AutoScaleBaseSize` property width and height are 1, and the control locations and sizes are used directly.

However, if the program runs on a machine with a Large Fonts setting, the font width and height are 6 and 15, respectively. When your form's constructor concludes its processing, Windows Forms scales the horizontal locations and sizes of all the controls by a factor of 6/5 (that's the average width associated with the form's `Font` property divided by `AutoScaleBaseSize.Width`). The vertical locations and sizes are scaled by a factor of 15/13, which is the height of the form's font divided by `AutoScaleBaseSize.Height`.

And that's how the Windows Forms Designer gets away with hard-coding pixel coordinates and sizes.

You can do something like this yourself. You can code all the locations and sizes of controls using values that work right for your machine and then insert a statement that sets the `AutoScaleBaseSize` property to a `Size` also appropriate for your machine. Then you'll want to check out the program with a different font size, either by changing your display settings or by setting a new `Font` property in the constructor.

However, you can also use `AutoScaleBaseSize` in more interesting ways.

**Creative `AutoScaleBaseSize` Settings**

Traditionally, Windows programmers coding in C and C++ using the Win32 API or the Microsoft Foundation Class (MFC) Library define their dialog boxes in a text format known as a `dialog box template` using a special device-independent coordinate system known as `dialog box coordinates`. All x coordinates are 1/4 of the average character width; y coordinates are expressed in units of 1/8 of
the character height. If a control's location is specified as (40, 32), for example, the control is positioned 10 average character widths from the left of the dialog box and 4 character heights from the top.

A Windows Forms program can use this same traditional dialog box coordinate system. All that's necessary to make it work is the following statement in the constructor:

```
AutoScaleBaseSize = new Size(4, 8);
```

Here's another alternative: you can even specify locations and sizes entirely in units of integral character heights and character widths. Then you only need the statement

```
AutoScaleBaseSize = new Size(1, 1);
```

You may find such a coordinate system just a bit too coarse, however. Usually at least half a character height resolution is necessary for attractively spacing controls vertically.

### Inside Auto-Scale

After the code in your form's constructor is executed, the form and all children in the form are scaled based on the form's `Font` and `AutoScaleBaseSize` properties. The actual scaling is performed by a protected method of the `Control` class named `ScaleCore`, which is called first for the form and then for all the controls that are children of the form.

You can accomplish the same scaling as auto-scale by calling one of the `Scale` methods for the form:

```
Control Methods (selection)

void Scale(float fScale)
void Scale(float xScale, yScale)
```

For example, if you set the `AutoScale` property to `false`, you can mimic auto-scaling by inserting the following statement at the end of the form's constructor:

```
Scale(GetAutoScaleSize(Font).Width / AutoScaleBaseSize.Width,
       GetAutoScaleSize(Font).Height / AutoScaleBaseSize.Height);
```

That's the width and height of the form's `Font` divided by the width and height you've specified in the `AutoScaleBaseSize` property.

If you need to rescale existing controls later on—probably because you change the `Font` property someplace other than in the constructor—you can't rely on auto-scale to do it for you. You need to call `Scale` directly. But be aware that the form and controls don't retain any previous scaling history. Once the controls are scaled following the constructor code, they have simple pixel locations and sizes. You can't call `Scale` again based on the form's `Font` and `AutoScaleBaseSize`. You'll need to calculate the scaling factors based on the size of the old font and the size of the new font.

Here's a program that creates five push buttons that let you select five different font sizes. The program's constructor uses locations and sizes based on the traditional dialog box coordinates. Auto-scaling handles the initial scaling. When you press a button, the `Click` event handler scales everything again based on the existing font and the new font.

### AutoScaleDemo.cs

```csharp
// AutoScaleDemo.cs © 2001 by Charles Petzold
//----------------------------------------------------------------------------
using System;
using System.Drawing;
```
using System.Windows.Forms;

class AutoScaleDemo: Form
{
    public static void Main()
    {
        Application.Run(new AutoScaleDemo());
    }

    public AutoScaleDemo()
    {
        Text = "Auto-Scale Demo";
        Font = new Font("Arial", 12);
        FormBorderStyle = FormBorderStyle.FixedSingle;
        int[] aiPointSize = { 8, 12, 16, 24, 32 };

        for (int i = 0; i < aiPointSize.Length; i++)
        {
            Button btn    = new Button();
            btn.Parent    = this;
            btn.Text      = "Use " + aiPointSize[i] + "-point font";
            btn.Tag       = aiPointSize[i];
            btn.Location  = new Point(4, 16 + 24 * i);
            btn.Size      = new Size(80, 16);
            btn.Click    += new EventHandler(ButtonOnClick);
        }
        ClientSize = new Size(88, 16 + 24 * aiPointSize.Length);
        AutoScaleBaseSize = new Size(4, 8);
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        pea.Graphics.DrawString(Text, Font, new SolidBrush(ForeColor), 0, 0);
    }

    void ButtonOnClick(object obj, EventArgs ea)
    {
        Button btn = (Button) obj;

        SizeF sizefOld = GetAutoScaleSize(Font);
        Font = new Font(Font.FontFamily, (int) btn.Tag);
        SizeF sizefNew = GetAutoScaleSize(Font);

        Scale(sizefNew.Width / sizefOld.Width,
              sizefNew.Height / sizefOld.Height);
    }
This program uses the *Tag* property of the *Button* control to store the integer point size associated with each button. During the *ButtonOnClick* method, the *Tag* property is cast to an integer for creating the font. Here's what the program looks like with the 12-point font set for the form:

As you press each button, the entire form is resized to reflect the new font size.

If you create a control somewhere other than in your form's constructor, you may need to use *Scale*. Call the new control's *Scale* method with the sizes of the current font and the *AutoSizeBaseSize* property.

**A Hexadecimal Calculator**

Here's a program that creates 29 *Button* controls to implement an infix notation hexadecimal calculator. The HexCalc program works with 64-bit unsigned integers and does addition, subtraction, multiplication, division, and remainders; bitwise AND, OR, and exclusive OR operations; and left and right bits shifts. Here's what the program looks like:
You can use either the mouse or the keyboard with HexCalc. You begin by "clicking in" or typing the number (up to 16 hexadecimal digits), then the operation, and then the second number. You can then show the result by clicking the Equals button or by pressing either the equals (=) or the Enter key. To correct your entries, click the button labeled Back or press the Backspace key. Click the display box, or press the Esc key to clear the current entry.

This is not a program I would care to put together in the Visual C# .NET Windows Forms Designer. This many buttons of identical coordinates and sizes cry out for a more methodical approach. After a couple false starts, I decided to subclass Button in a class named CalcButton. In the CalcButton class, I implemented a constructor that has arguments for the button's parent, its text, its location and size, and an additional field named chKey that contains the keyboard character that invokes the button. The HexCalc constructor contains 29 new CalcButton statements that create all the buttons. I use traditional dialog box coordinates for the buttons, but I call the short version of Scale directly to scale equally in all directions. That preserves the square appearance of most of the buttons.

HexCalc.cs
//------------------------
// HexCalc.cs © 2001 by Charles Petzold
//------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HexCalc: Form
{
    Button btnResult;
    ulong ulNum = 0;
    ulong ulFirstNum = 0;
    bool bNewNumber = true;
    char chOperation = '=';

    public static void Main()
    {
        Application.Run(new HexCalc());
    }
    public HexCalc()
    {
Text = "Hex Calc";
Icon = new Icon(GetType(), "HexCalc.HexCalc.ico");
FormBorderStyle = FormBorderStyle.FixedSingle;
MaximizeBox = false;

new CalcButton(this, "D", 'D', 8, 24, 14, 14);
new CalcButton(this, "A", 'A', 8, 40, 14, 14);
new CalcButton(this, "7", '7', 8, 56, 14, 14);
new CalcButton(this, "4", '4', 8, 72, 14, 14);
new CalcButton(this, "1", '1', 8, 88, 14, 14);
new CalcButton(this, "0", '0', 8, 104, 14, 14);
new CalcButton(this, "E", 'E', 26, 24, 14, 14);
new CalcButton(this, "B", 'B', 26, 40, 14, 14);
new CalcButton(this, "8", '8', 26, 56, 14, 14);
new CalcButton(this, "5", '5', 26, 72, 14, 14);
new CalcButton(this, "2", '2', 26, 88, 14, 14);
new CalcButton(this, "Back", '\x08', 26, 104, 32, 14);
new CalcButton(this, "C", 'C', 44, 40, 14, 14);
new CalcButton(this, "F", 'F', 44, 24, 14, 14);
new CalcButton(this, "9", '9', 44, 56, 14, 14);
new CalcButton(this, "6", '6', 44, 72, 14, 14);
new CalcButton(this, "3", '3', 44, 88, 14, 14);
new CalcButton(this, "+", '+', 62, 24, 14, 14);
new CalcButton(this, "-", '-', 62, 40, 14, 14);
new CalcButton(this, "*", '*', 62, 56, 14, 14);
new CalcButton(this, "/", '/', 62, 72, 14, 14);
new CalcButton(this, "%", '%', 62, 88, 14, 14);
new CalcButton(this, "Equals", '=', 62, 104, 32, 14);
new CalcButton(this, "&amp;", '&', 80, 24, 14, 14);
new CalcButton(this, "|", '|', 80, 40, 14, 14);
new CalcButton(this, "^", '^', 80, 56, 14, 14);
new CalcButton(this, "<", '<', 80, 72, 14, 14);
new CalcButton(this, ">", '>', 80, 88, 14, 14);

btnResult =
    new CalcButton(this, "0", '\x1B', 8, 4, 86, 14);

foreach (Button btn in Controls)
    btn.Click += new EventHandler(ButtonOnClick);

ClientSize = new Size(102, 126);
Scale(Font.Height / 8f);
}
protected override void OnKeyPress(KeyPressEventArgs kpea)
{

char chKey = Char.ToUpper(kpea.KeyChar);

if (chKey == '\x0D')
    chKey = '=';

for (int i = 0; i < Controls.Count; i++)
{
    CalcButton btn = (CalcButton) Controls[i];

    if (chKey == btn.chKey)
    {
        InvokeOnClick(btn, EventArgs.Empty);
        break;
    }
}
}

void ButtonOnClick(object obj, EventArgs ea)
{
    CalcButton btn = (CalcButton) obj;

    if (btn.chKey == '\x08')
        ulNum /= 16;

    else if (btn.chKey == '\x1B')
        ulNum = 0;

    else if (Char.IsLetterOrDigit(btn.chKey))  // Hex digit
    {
        if (bNewNumber)
        {
            ulFirstNum = ulNum;
            ulNum = 0;
            bNewNumber = false;
        }

        if (ulNum <= ulong.MaxValue >> 4)
            ulNum = 16 * ulNum +
                    (ulong)(btn.chKey -
                           (Char.IsDigit(btn.chKey) ? '0' : 'A' - 10));
    }
    else  // Operation
    {
        if (!bNewNumber)
        {
            switch(chOperation)
{  
case '=': ulNum = ulNum; break;
case '+': ulNum = ulFirstNum + ulNum; break;
case '-': ulNum = ulFirstNum - ulNum; break;
case '*': ulNum = ulFirstNum * ulNum; break;
case '&': ulNum = ulFirstNum & ulNum; break;
case '|': ulNum = ulFirstNum | ulNum; break;
case '^': ulNum = ulFirstNum ^ ulNum; break;
case '<': ulNum = ulFirstNum << (int)ulNum; break;
case '>': ulNum = ulFirstNum >> (int)ulNum; break;
case '/': ulNum = ulNum != 0 ? ulFirstNum / ulNum : ulong.MaxValue;
break;
case '%': ulNum = ulNum != 0 ? ulFirstNum % ulNum : ulong.MaxValue;
break;
default: ulNum = 0; break;
}

bNewNumber = true;
chOperation = btn.chKey;
}
btnResult.Text = String.Format("{0:X}", ulNum);
}
}

class CalcButton: Button
{
  public char chKey;

  public CalcButton(Control parent, string str, char chkey, 
                     int x, int y, int cx, int cy)
  {
    Parent = parent;
    Text = str;
    chKey = chkey;
    Location = new Point(x, y);
    Size = new Size(cx, cy);
    SetStyle(ControlStylesSelectable, false);
  }
}

HexCalc.ico
The tricky part of this program was the keyboard interface. I didn't want the keys themselves to get
the input focus. The dotted outline that the button draws to indicate input focus just didn't look right in
this program. Shifting focus among the buttons by using the Tab key didn't make much sense either.
And I had more keyboard equivalents than buttons.

To force keyboard events to the form, each button sets its Selectable style to false. This style
prevents the button from obtaining the input focus.

The OnKeyPress method loops through the Controls array and finds the button corresponding to the
keystroke. It then calls InvokeOnClick to mimic a Click event for the button. The ButtonOnClick
method thus handles both button mouse clicks and keyboard equivalents.

Radio Buttons and Group Boxes

Someday, no one will know why they're called radio buttons. You see, car radios once came
equipped with a row of tall buttons that could be set to favorite radio stations. To select a station, you
pushed in a button, which caused the previously pushed-in button to pop out. Because only one
button could be pressed at a time, a group of radio button controls always reflects a group of
mutually exclusive options.

What makes radio buttons different from other controls is that they always exist in a group. Because
one (and only one) button in a group can be checked at any time, the states of the radio buttons
affect each other. Turning one radio button on turns another off. The keyboard navigation is also
somewhat different. Within a group of radio buttons, the arrow keys are supposed to move the input
focus from button to button. As the input focus changes, the checked radio button also changes. The
Tab key is supposed to move from the group of radio buttons to the next control. When you use the
Tab key to move into a group of radio buttons, the checked radio button receives the input focus.

Fortunately, much of this user interface is taken care of for you. For each group of radio buttons, all
you need to do is create a control of type GroupBox and make the GroupBox a child of your form.
Then you make all the RadioButton objects in the group children of the GroupBox.

Let's look at the RadioButton class first. Like CheckBox, the RadioButton class includes properties
named Checked and AutoCheck:

<table>
<thead>
<tr>
<th>RadioButton Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

The Checked property indicates whether the radio button is checked (which it visually illustrates by
displaying a solid dot in a circle). Keeping the AutoCheck property set to true automates the process
of using radio buttons. As the user clicks the radio buttons (or moves the input focus among the
group), the radio buttons are automatically checked and unchecked. If you set AutoCheck to false,
you'll have to install Click event handlers and do all the checking and unchecking yourself.

The only other public properties that RadioButton defines are Appearance and CheckAlign, which
work just as they do in the CheckBox class.
The *CheckedChanged* event occurs whenever a radio button is checked or unchecked, either by the user or by the program:

**RadioButton Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckedChanged</td>
<td>OnCheckedChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

You'll get two *CheckedChanged* events in a row, one for the radio button being unchecked and then one for the radio button being checked. You can tell these two events apart by looking at the *Checked* property of the button. During the *CheckedChanged* event, the *Checked* property reflects the new state.

The *GroupBox* class is a descendent of *Control* but implements only one public property (*FlatStyle*) and no additional methods or events beyond what *Control* defines.

Let's look at an example. The following program draws an ellipse based on the setting of eight radio buttons and one check box.

**RadioButtons.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class RadioButtons: Form
{
    bool bFillEllipse;
    Color colorEllipse;

    static void Main()
    {
        Application.Run(new RadioButtons());
    }

    RadioButtons()
    {
        Text = "Radio Buttons Demo";
        ResizeRedraw = true;

        string[] astrColor = { "Black", "Blue", "Green", "Cyan",
            "Red", "Magenta", "Yellow", "White"};

        GroupBox grpbox = new GroupBox();
        grpbox.Parent = this;
        grpbox.Text   = "Color";
        grpbox.Location = new Point(Font.Height / 2, Font.Height / 2);
        grpbox.Size   = new Size(9 * Font.Height,
            (3 * astrColor.Length + 4) * Font.Height / 2);
```
for (int i = 0; i < astrColor.Length; i++)
{
    RadioButton radiobtn = new RadioButton();
    radiobtn.Parent = grpbox;
    radiobtn.Text = astrColor[i];
    radiobtn.Location = new Point(Font.Height, 3 * (i + 1) * Font.Height / 2);
    radiobtn.Size = new Size(7 * Font.Height, 3 * Font.Height / 2);
    radiobtn.CheckedChanged +=
    new EventHandler(RadioButtonOnCheckedChanged);
    if (i == 0)
        radiobtn.Checked = true;
}
CheckBox chkbox = new CheckBox();
chkbox.Parent = this;
chkbox.Text = "Fill Ellipse";
chkbox.Location = new Point(Font.Height, 3 * (astrColor.Length + 2) * Font.Height / 2);
chkbox.Size = new Size(Font.Height * 7, 3 * Font.Height / 2);
chkbox.CheckedChanged +=
    new EventHandler(CheckBoxOnCheckedChanged);
}
void RadioButtonOnCheckedChanged(object obj, EventArgs ea)
{
    RadioButton radiobtn = (RadioButton) obj;

    if (radiobtn.Checked)
    {
        colorEllipse = Color.FromName(radiobtn.Text);
        Invalidate(false);
    }
}
void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    bFillEllipse = ((CheckBox)obj).Checked;
    Invalidate(false);
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
Rectangle rect = new Rectangle(10 * Font.Height, 0, 
ClientSize.Width - 
10 * Font.Height - 1, 
ClientSize.Height - 1);

if(bFillEllipse)
    grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
else
    grfx.DrawEllipse(new Pen(colorEllipse), rect);
}
}

An array of eight colors is defined toward the beginning of the constructor. All the vertical coordinates and sizes the program calculates are generalized enough to accommodate additional colors in this array, just as long as you make sure they're actual .NET Framework color names. (The width of the controls isn't sufficient to accommodate some of the longer color names, however.)

The constructor first creates a GroupBox control. The parent of the group box is the form. Next, the constructor creates eight radio buttons that are children of the group box. Notice at the bottom of the for loop that the program sets the Checked property of the first radio button. That statement generates a call to RadioButtonOnCheckedChanged, which initializes the colorEllipse field. The constructor concludes by creating a CheckBox control as a child of the form.

You can test that the keyboard interface works as I described. As you use the arrow keys to move the focus among the radio buttons, the buttons make calls to RadioButtonOnCheckedChanged. That method uses the static Color.FromName method to convert the button text to a Color object. Both this method and CheckBoxOnCheckedChanged invalidate the client area to generate a call to OnPaint, which paints the ellipse:

![Radio Buttons Demo](image)

**Scroll Bars**

When the subject of scroll bars first came up in Chapter 4, I discussed some of the differences between scroll bar controls and the scroll bars created as part of the auto-scroll feature in any class descended from ScrollableControl (including Form and Panel). With the auto-scroll feature, you specify the size of the client area you want, and the scroll bars appear automatically at the bottom and right of the client area. The auto-scroll scroll bars have no events associated with them—at least none that an application can get access to.

The ScrollBar class is an abstract class descended from Control.
Descended from `ScrollBar` are the horizontal scroll bar (`HScrollBar`) and the vertical scroll bar (`VScrollBar`). You can position these scroll bar controls anywhere in your client area, and even make them whatever size you want. Although horizontal scroll bars have a default height and vertical scroll bars have a default width, you can indeed make very thin scroll bars or very pudgy ones. However, you can't set the background color or foreground color of scroll bars.

To keep the terminology consistent between horizontal and vertical scroll bars, let me refer to `thickness` and `length`. Thickness is the height of horizontal scroll bars and the width of vertical scroll bars. Length is the width of horizontal scroll bars and the height of vertical scroll bars. By default, newly created scroll bars have their thickness set to standard values—the same values you can obtain from `SystemInformation.VerticalScrollBarWidth` and `SystemInformation.HorizontalScrollBarHeight`.

Here are the five main properties that the `ScrollBar` class adds to `Control`:

**ScrollBar Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Value</td>
<td>get/set</td>
<td>Ranges from Minimum to (Maximum + 1 − LargeChange)</td>
</tr>
<tr>
<td>int</td>
<td>Minimum</td>
<td>get/set</td>
<td>Default is 0</td>
</tr>
<tr>
<td>int</td>
<td>Maximum</td>
<td>get/set</td>
<td>Default is 100</td>
</tr>
<tr>
<td>int</td>
<td>SmallChange</td>
<td>get/set</td>
<td>Default is 1</td>
</tr>
<tr>
<td>int</td>
<td>LargeChange</td>
<td>get/set</td>
<td>Default is 10</td>
</tr>
</tbody>
</table>

The `Value` property indicates the position of the scroll box on the scroll bar. It ranges from the `Minimum` setting to the well, not quite `Maximum` setting. If you click the arrows at the ends of the scroll bar, the `Value` property changes by `SmallChange`. If you click on either side of the scroll box, the `Value` property changes by `LargeChange`.

Why does the `Value` range from `Minimum` to `(Maximum + 1 − LargeChange)`? Think of a document, perhaps a word processing document, that contains 500 lines of text. You set `Minimum` to 0 and `Maximum` to 499. Your client area is large enough to display 25 lines of text. Set `SmallChange` to 1 (that is, one line of text) and `LargeChange` to 25.

The size of the scroll box relative to the length of the scroll bar is based on the ratio of `LargeChange` to `Maximum`. That's the proportion of the document you can view.

When `Value` is 0, you view the top of the document, which comprises—assuming you're using zero-based indexing of the lines of the document—lines 0 through 24. When `Value` is 1, you view lines 1 through 25. And when `Value` is 475, you view lines 475 through 499. That's the bottom of the document, which means that `Value` doesn't need to get any higher. And that's why `Value` doesn't get higher than `(Maximum + 1 − LargeChange)`.

If you're not dealing with a document, you need to set `Maximum` so that you get the correct range for `Value`. I'll have an example soon.
Two events are implemented by ScrollBar:

**ScrollBar Events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Handler</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueChanged</td>
<td>OnValueChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Scroll</td>
<td>OnScroll</td>
<td>ScrollEventHandler</td>
<td>ScrollEventArgs</td>
</tr>
</tbody>
</table>

The `ValueChanged` event occurs only when the `Value` property really truly changes. If the cat lies down on your keyboard, `ValueChanged` won’t waste your time with a bunch of superfluous events.

The `ValueChanged` event occurs not only when the user manipulates the scroll bar but also when the program sets the `Value` property. The `Scroll` event doesn’t occur when the `Value` property is programmatically changed.

Moreover, the `Scroll` event gives you much more information about how the scroll bar is being manipulated. It’s possible you might never need to use the `Scroll` event, but it’s there if you find `ValueChanged` insufficient. The event handler for the `Scroll` event gets an object of type `ScrollEventArgs`, which has the following properties:

**ScrollEventArgs Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>NewValue</td>
<td>get/set</td>
</tr>
<tr>
<td>ScrollEventType</td>
<td>Type</td>
<td>get</td>
</tr>
</tbody>
</table>

The `NewValue` property is what the scroll bar `Value` property will be set to after the event handler returns control back to the scroll bar. You can override that property by setting `NewValue` to something else. The `Type` property is of type `ScrollEventType`.

**ScrollEventType Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| SmallDecrement| 0     | Mouse: Left or top arrow  
Keyboard: Left or Up arrow |
| SmallIncrement| 1     | Mouse: Right or bottom arrow  
Keyboard: Right or Down arrow |
| LargeDecrement| 2     | Mouse: Left or top area  
Keyboard: Page Up |
| LargeIncrement| 3     | Mouse: Right or bottom area  
Keyboard: Page Down |
| ThumbPosition | 4     | Mouse: Button up on scroll box (thumb) |
| ThumbTrack    | 5     | Mouse: Button down on scroll box (thumb) or move |
| First         | 6     | Keyboard: Home |
| Last          | 7     | Keyboard: End |
| EndScroll     | 8     | Scrolling operation completed |

For example, suppose a scroll bar has the input focus and you press and release one of the keyboard arrow keys. Or you click with the mouse on the scroll bar arrow. You’ll first get a `Scroll` event with the `Type` field set to `ScrollEventArgsType.SmallIncrement` or `ScrollEventArgsType.SmallDecrement`. Then you’ll receive a `ValueChanged` event, followed by another `Scroll` event with the `Type` field equal to `ScrollEventArgsType.EndScroll`. If `sb` is an object of type `ScrollBar` and `sea` is an object of type `ScrollEventArgs`, here’s the sequence of events:

<table>
<thead>
<tr>
<th>Event</th>
<th><code>sb.Value</code></th>
<th><code>sea.Type</code></th>
<th><code>sea.NewValue</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scroll</td>
<td>N</td>
<td>SmallIncrement</td>
<td>N + 1</td>
</tr>
</tbody>
</table>
If you keep the mouse button (or the arrow key) pressed, you'll get a series of events, finally terminated with an `EndScroll`:

<table>
<thead>
<tr>
<th>Event</th>
<th>sb.Value</th>
<th>sea.Type</th>
<th>sea.NewValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueChanged</td>
<td>$N + 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>$N + 1$</td>
<td><code>EndScroll</code></td>
<td>$N + 1$</td>
</tr>
</tbody>
</table>

You won't get a `ValueChanged` event if the `Value` has reached its minimum or maximum. If you press the End key, generally you'll get the following:

<table>
<thead>
<tr>
<th>Event</th>
<th>sb.Value</th>
<th>sea.Type</th>
<th>sea.NewValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scroll</td>
<td>$N$</td>
<td><code>Last</code></td>
<td>$sb.Maximum$</td>
</tr>
<tr>
<td>ValueChanged</td>
<td>$sb.Maximum$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>$sb.Maximum$</td>
<td><code>EndScroll</code></td>
<td>$sb.Maximum$</td>
</tr>
</tbody>
</table>

However, if the scroll box is already at the end of the scroll bar, when you press the End key, you'll get the following:

<table>
<thead>
<tr>
<th>Event</th>
<th>sb.Value</th>
<th>sea.Type</th>
<th>sea.NewValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scroll</td>
<td>$sb.Maximum$</td>
<td><code>Last</code></td>
<td>$sb.Maximum$</td>
</tr>
<tr>
<td>Scroll</td>
<td>$sb.Maximum$</td>
<td><code>EndScroll</code></td>
<td>$sb.Maximum$</td>
</tr>
</tbody>
</table>

If you grab the scroll box with the mouse and move it, you get the following sequence of events:

<table>
<thead>
<tr>
<th>Event</th>
<th>sb.Value</th>
<th>sea.Type</th>
<th>sea.NewValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scroll</td>
<td>$N$</td>
<td><code>ThumbTrack</code></td>
<td>$N$</td>
</tr>
<tr>
<td>Scroll</td>
<td>$N$</td>
<td><code>ThumbTrack</code></td>
<td>$N + 1$</td>
</tr>
<tr>
<td>ValueChanged</td>
<td>$N + 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>$N + 1$</td>
<td><code>ThumbTrack</code></td>
<td>$N + 2$</td>
</tr>
<tr>
<td>ValueChanged</td>
<td>$N + 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>$N + 2$</td>
<td><code>ThumbTrack</code></td>
<td>$N + 3$</td>
</tr>
<tr>
<td>ValueChanged</td>
<td>$N + 3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scroll</td>
<td>$N + 3$</td>
<td><code>ThumbTrack</code></td>
<td>$N + 4$</td>
</tr>
<tr>
<td>ValueChanged</td>
<td>$N + 4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Depending on how fast you move the scroll bar, you might not get events for every possible value. And it’s really how your program reacts to quick movement of the scroll box that determines whether you should install a Scroll handler rather than a ValueChanged handler. Try grabbing the scroll box with the mouse and shaking it violently. If your program can’t keep up, consider the possibility of processing the Scroll event rather than ValueChanged. You can then ignore all values of Type except EndScroll, for example.

The ColorScroll program uses three scroll bars, labeled Red, Green, and Blue, that let you select a color mix. The program sets the form’s background color to the resultant color you select. To keep all the scroll bars and labels visible, a white Panel control covers half the client area of the form. All the other controls—three scroll bars and six labels—are children of Panel.

```
ColorScroll.cs
//--------------------------------------------------
// ColorScroll.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorScroll: Form
{
    Panel panel;
    Label[] alabelName = new Label[3];
    Label[] alabelValue = new Label[3];
    VScrollBar[] avscroll = new VScrollBar[3];

    public static void Main()
    {
        Application.Run(new ColorScroll());
    }
    public ColorScroll()
    {
        Text = "Color Scroll";
        Color[] acolor = { Color.Red, Color.Green, Color.Blue };

        // Create the panel.

        panel = new Panel();
        panel.Parent = this;
        panel.Location = new Point(0, 0);
        panel.BackColor = Color.White;

        // Loop through the three colors.
```
for (int i = 0; i < 3; i++)
{
    alabelName[i] = new Label();
    alabelName[i].Parent = panel;
    alabelName[i].ForeColor = acolor[i];
    alabelName[i].Text = "&" + acolor[i].ToKnownColor();
    alabelName[i].TextAlign = ContentAlignment.MiddleCenter;

    avscroll[i] = new VScrollBar();
    avscroll[i].Parent = panel;
    avscroll[i].SmallChange = 1;
    avscroll[i].LargeChange = 16;
    avscroll[i].Minimum  = 0;
    avscroll[i].Maximum = 255 + avscroll[i].LargeChange - 1;
    avscroll[i].ValueChanged +=
        new EventHandler(ScrollOnValueChanged);
    avscroll[i]._TabStop = true;

    alabelValue[i] = new Label();
    alabelValue[i].Parent = panel;
    alabelValue[i].TextAlign = ContentAlignment.MiddleCenter;
}

Color color = BackColor;
avscroll[0].Value = color.R;  // Generates ValueChanged event
avscroll[1].Value = color.G;
avscroll[2].Value = color.B;

OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    int cx = ClientSize.Width;
    int cy = ClientSize.Height;
    int cyFont = Font.Height;

    panel.Size = new Size(cx / 2, cy);

    for (int i = 0; i < 3; i++)
    {
        alabelName[i].Location = new Point(i * cx / 6, cyFont / 2);
        alabelName[i].Size = new Size(cx / 6, cyFont);
The constructor creates all the controls and stores them as fields. The scroll bars must provide values from 0 through 255. Notice how I set `LargeChange` to 16 and then set the `Maximum` property to 255 plus `LargeChange` minus 1, which equals 270. The constructor doesn’t position or size the controls, however. That brutal job is the responsibility of the `OnResize` method. The location and sizes are based on the size of the client area and the font height. The vertical scroll bars change width as you resize the form. (I tried to use anchoring for the effect I wanted, but I just couldn’t get it to work right.) Here’s a normal-size view of the program:

![Color Scroll](image)

There are two sets of labels: the three Label controls stored in the `alabelName` array are assigned the `Text` properties `Red`, `Green`, and `Blue`, and get their `ForeColor` properties set to the same color. I use the `acolor` array for both jobs. If you use the `ToString` method with a `Color` object, you get something like `Color [Red]`. But if the `Color` object is part of the `KnownColor` enumeration, you can convert the `Color` object to a `KnownColor` value by using the method `ToKnownColor`. The enumeration value converts to a string like `Red`. 

```csharp
avscroll[i].Location = new Point((4 * i + 1) * cx / 24,
    2 * cyFont);
avscroll[i].Size = new Size(cx / 12, cy - 4 * cyFont);

alabelValue[i].Location = new Point(i * cx / 6,
    cy - 3 * cyFont / 2);
alabelValue[i].Size = new Size(cx / 6, cyFont);
}
}

void ScrollOnValueChanged(Object obj, EventArgs ea)
{
    for (int i = 0; i < 3; i++)
        if ((VScrollBar) obj == avscroll[i])
            alabelValue[i].Text = avscroll[i].Value.ToString();

    BackColor = Color.FromArgb(avscroll[0].Value,
        avscroll[1].Value,
        avscroll[2].Value);
}

The constructor creates all the controls and stores them as fields. The scroll bars must provide values from 0 through 255. Notice how I set `LargeChange` to 16 and then set the `Maximum` property to 255 plus `LargeChange` minus 1, which equals 270. The constructor doesn’t position or size the controls, however. That brutal job is the responsibility of the `OnResize` method. The location and sizes are based on the size of the client area and the font height. The vertical scroll bars change width as you resize the form. (I tried to use anchoring for the effect I wanted, but I just couldn’t get it to work right.) Here’s a normal-size view of the program:

There are two sets of labels: the three Label controls stored in the `alabelName` array are assigned the `Text` properties `Red`, `Green`, and `Blue`, and get their `ForeColor` properties set to the same color. I use the `acolor` array for both jobs. If you use the `ToString` method with a `Color` object, you get something like `Color [Red]`. But if the `Color` object is part of the `KnownColor` enumeration, you can convert the `Color` object to a `KnownColor` value by using the method `ToKnownColor`. The enumeration value converts to a string like `Red`. 

```csharp
avscroll[i].Location = new Point((4 * i + 1) * cx / 24,
    2 * cyFont);
avscroll[i].Size = new Size(cx / 12, cy - 4 * cyFont);

alabelValue[i].Location = new Point(i * cx / 6,
    cy - 3 * cyFont / 2);
alabelValue[i].Size = new Size(cx / 6, cyFont);
}
}

void ScrollOnValueChanged(Object obj, EventArgs ea)
{
    for (int i = 0; i < 3; i++)
        if ((VScrollBar) obj == avscroll[i])
            alabelValue[i].Text = avscroll[i].Value.ToString();

    BackColor = Color.FromArgb(avscroll[0].Value,
        avscroll[1].Value,
        avscroll[2].Value);
}

```
The TabStop property inherited from Control is normally set to false for scroll bars. ColorScroll sets it to true. In addition, the Red, Green, and Blue labels are prefaced with an ampersand. Because labels are not tab stops, if they contain a letter preceded by an ampersand, the letter functions as an accelerator in setting the input to the next tab stop control. So not only can you shift the input focus among the scroll bars using the Tab key, you can also shift input focus by pressing R, G, or B.

When you move one of the scroll bars, it generates a ValueChanged event and a call to the program's ScrollOnValueChanged method. This method casts the obj argument to a VScrollBar object and then searches through the avscroll array to find the match. The resultant index is used to set the corresponding Label control below the scroll bar that displays the value (alabelValue). The method concludes by using the values from all three scroll bars to recompute a background color for the form.

Watch out when setting the scroll bar Value property from your program! The constructor for ColorScroll originally concluded with the following three statements to initialize the three scroll bars with the background color of the form:

```csharp
avscroll[0].Value = BackColor.R;
avscroll[1].Value = BackColor.G;
avscroll[2].Value = BackColor.B;
```

However, the first statement caused a ValueChanged event, which performed a call to ScrollOnValueChanged in the program, which then set the background color based on the three scroll bar Value properties. But because the Green and Blue scroll bars hadn't been initialized yet, the background color effectively had its green and blue components—BackColor.G and BackColor.B—set to 0. Saving the background color first in another Color variable and using that variable to set the Value properties fixed the problem:

```csharp
Color color = BackColor;
avscroll[0].Value = color.R;
avscroll[1].Value = color.G;
avscroll[2].Value = color.B;
```

The Track Bar Alternative

Very similar in functionality to scroll bars are track bars. From the programmer's perspective, one difference between scroll bars and track bars is that the horizontal or vertical orientation of a track bar is a property:

<table>
<thead>
<tr>
<th>TrackBar Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Orientation</td>
</tr>
</tbody>
</table>

The Orientation enumeration is short and simple:

<table>
<thead>
<tr>
<th>Orientation Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Horizontal</td>
</tr>
<tr>
<td>Vertical</td>
</tr>
</tbody>
</table>

As you know from experimenting with the ColorScroll program, you can change the thickness of scroll bars. By default, you can't change the thickness of track bars, and the track bar is less amenable to such changes. The track bar usually needs a minimum thickness to display the tick marks, and it doesn't really use any extra thickness. If you want to experiment with changing the thickness of track bars, you must set the AutoSize property to false:

<table>
<thead>
<tr>
<th>TrackBar Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>
By default, the `AutoSize` property is `true`, which means that the track bar will have a constant width (for vertical track bars) or height (for horizontal track bars) regardless of the `Size` property. The default `TabStop` property for track bars is also set to `true` (unlike scroll bars).

The `TrackBar` class has the following same properties as `ScrollBar` but with different `Maximum` and `LargeChange` defaults:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Value</td>
<td>get/set</td>
<td>Ranges from <code>Minimum</code> to <code>Maximum</code></td>
</tr>
<tr>
<td>int</td>
<td>Minimum</td>
<td>get/set</td>
<td>Default is 0</td>
</tr>
<tr>
<td>int</td>
<td>Maximum</td>
<td>get/set</td>
<td>Default is 10</td>
</tr>
<tr>
<td>int</td>
<td>SmallChange</td>
<td>get/set</td>
<td>Default is 1</td>
</tr>
<tr>
<td>int</td>
<td>LargeChange</td>
<td>get/set</td>
<td>Default is 5</td>
</tr>
</tbody>
</table>

The `Value` property of track bars ranges from `Minimum` to `Maximum` without any trickiness involving the `LargeChange` property. This actually makes track bars easier to use for applications like ColorScroll but harder to use for applications in which a document is scrolled.

Although vertical scroll bars have increasing values as you scroll the scroll box down, vertical track bars have increasing values as you scroll the scroll box up. Again, it's the difference between scrolling a document and selecting a value.

Two additional properties let you control tick marks on the track bar:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TickStyle</td>
<td>TickStyle</td>
<td>get/set</td>
<td>Default is <code>BottomRight</code></td>
</tr>
<tr>
<td>int</td>
<td>TickFrequency</td>
<td>get/set</td>
<td>Default is 1</td>
</tr>
</tbody>
</table>

The `TickStyle` property lets you specify which side of the track bar contains the tick marks based on the following enumeration:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>No tick marks</td>
</tr>
<tr>
<td>TopLeft</td>
<td>1</td>
<td>Tick marks on top for horizontal track bars and on left for vertical track bars</td>
</tr>
<tr>
<td>BottomRight</td>
<td>2</td>
<td>Tick marks on bottom for horizontal track bars and on right for vertical track bars</td>
</tr>
<tr>
<td>Both</td>
<td>3</td>
<td>Tick marks on both sides</td>
</tr>
</tbody>
</table>

The default is `BottomRight`. If your `TickFrequency` is 1 (the default) and you set a wide range for the track bar, the tick marks may end up looking like a solid block of black.

You also have a bit more flexibility with track bars in specifying a background color or image:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>BackColor</td>
<td>get/set</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>BackgroundImage</td>
<td>get/set</td>
<td></td>
</tr>
</tbody>
</table>

The two crucial `TrackBar` events have the same names as those implemented in `ScrollBar`.
### TrackBar Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueChanged</td>
<td>OnValueChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Scroll</td>
<td>OnScroll</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

Both events are associated with normal `EventHandler` delegates. For track bars, the `Scroll` events and `ValueChanged` events always come in pairs (`Scroll` first, then `ValueChanged`) except when the `Value` property is programmatically set to a different value. In that case, a `ValueChanged` event occurs without a corresponding `Scroll` event.

Here's the ColorScroll program rewritten to use track bars.

**ColorTrackBar.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorTrackBar : Form
{
    Panel    panel;
    Label[]  alabelName  = new Label[3];
    Label[]  alabelValue = new Label[3];
    TrackBar[] atrackbar = new TrackBar[3];

    public static void Main()
    {
        Application.Run(new ColorTrackBar());
    }

    public ColorTrackBar()
    {
        Text = "Color Track Bar";

        Color[] acolor = { Color.Red, Color.Green, Color.Blue };

        // Create the panel.

        panel = new Panel();
        panel.Parent = this;
        panel.Location = new Point(0, 0);
        panel.BackColor = Color.White;

        // Loop through the three colors.

        for (int i = 0; i < 3; i++)
        {
            // Set the track bar properties.
            atrackbar[i].Min = 0;
            atrackbar[i].Max = 255;
            atrackbar[i].TickFrequency = 10;
            atrackbar[i].Value = 128;

            // Add a label to the panel.
            alabelName[i].Text = "Name: " + acolor[i].Name;
            alabelValue[i].Text = "Value: " + atrackbar[i].Value;
            panel.Controls.Add(atrackbar[i]);
            panel.Controls.Add(alabelName[i]);
            panel.Controls.Add(alabelValue[i]);
        }
    }
}
```

{  
    alabelName[i] = new Label();
    alabelName[i].Parent = panel;
    alabelName[i].ForeColor = acolor[i];
    alabelName[i].Text = "&" + acolor[i].ToKnownColor();
    alabelName[i].TextAlign = ContentAlignment.MiddleCenter;

    atrackbar[i] = new TrackBar();
    atrackbar[i].Parent = panel;
    atrackbar[i].Orientation = Orientation.Vertical;
    atrackbar[i].BackColor = acolor[i];
    atrackbar[i].SmallChange = 1;
    atrackbar[i].LargeChange = 16;
    atrackbar[i].Minimum = 0;
    atrackbar[i].Maximum = 255;
    atrackbar[i].TickFrequency = 16;
    atrackbar[i].ValueChanged +=
        new EventHandler(TrackBarOnValueChanged);

    alabelValue[i] = new Label();
    alabelValue[i].Parent = panel;
    alabelValue[i].TextAlign = ContentAlignment.MiddleCenter;
}

Color color = BackColor;
atrackbar[0].Value = color.R; // Generates ValueChanged event
atrackbar[1].Value = color.G;
atrackbar[2].Value = color.B;

OnResize(EventArgs.Empty);
}

protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    int cx = ClientSize.Width;
    int cy = ClientSize.Height;
    int cyFont = Font.Height;

    panel.Size = new Size(cx / 2, cy);

    for (int i = 0; i < 3; i++)
    {
        alabelName[i].Location = new Point(i * cx / 6, cyFont / 2);
        alabelName[i].Size = new Size(cx / 6, cyFont);
    }
at(bar[i].Height = cy - 4 * cyFont;
at(bar[i].Location =
    new Point((1 + 2 * i) * cx / 12 - at(bar[i].Width / 2,
            2 * cyFont);

al(labelValue[i].Location = new Point(i * cx / 6,
            cy - 3 * cyFont / 2);
    al(labelValue[i].Size = new Size(cx / 6, cyFont);
}
}
void TrackBarOnValueChanged(object obj, EventArgs ea)
{
    for (int i = 0; i < 3; i++)
        if((TrackBar) obj == at(bar[i])
            alabelValue[i].Text = at(bar[i].Value.ToString();

        BackColor = Color.FromArgb(at(bar[0].Value,
            at(bar[1].Value,
            at(bar[2].Value);
}
}
As you can see, there aren't many differences between the two programs. The code that sets the TrackBar properties mostly reflects the difference between scroll bars and track bars. The ColorTrackBar program doesn't need to set the TabStop property, but it does need to set the Orientation and TickFrequency properties. In addition, the program takes advantage of the fact that track bars color their backgrounds by setting the BackColor property to red, green, or blue. The OnResize method is a little different as well because I decided to let the track bars retain their default width.
Chapter 13: Béziers and Other Splines

Overview

What is a spline? Even recent dictionaries define spline as "a flexible piece of wood, hard rubber, or metal used in drawing curves." The definition conjures up a quaint image of an engineer wielding an awkward bendable contraption while hunched over a spread of graph paper attempting to fit a curve through a scattering of data points. These days a spline is more accurately described as "a curve calculated by a mathematical function that connects separate points with a high degree of smoothness.… See also Bézier curve."

Pierre Etienne Bézier was born in Paris in 1910 into a family of engineers. He received a degree in mechanical engineering in 1930 and a second degree in electrical engineering the following year. In 1933, he began working at the French automotive company Renault, where he remained until 1975. During the 1950s, Bézier was responsible for implementing some of the first drilling and milling machines that operated under NC, that is, numerical control (a term rarely used these days).

Beginning in 1960, much of Bézier's work was centered around the UNISURF program, an early CAD/CAM system used at Renault for interactively designing automobile parts. What was required in such a system were mathematical definitions of complex curves that designers could manipulate without knowing about the underlying mathematics, which could then be used in manufacturing processes. From this work came the curve that now bears Bézier's name. Pierre Bézier died in 1999.

The Bézier spline has come to assume a high degree of importance in computer graphics, ranking just under the straight line and the elliptical arc. In PostScript, the Bézier spline is used to render all curves—even elliptical arcs are approximated from Béziers. Bézier splines are also used to define the outlines of PostScript fonts. (TrueType uses a simpler and faster form of spline.)

The Bézier Spline in Practice

A single Bézier spline is uniquely defined by four points, which we can call $p_0$, $p_1$, $p_2$, and $p_3$. The curve begins at $p_0$ and ends at $p_3$; thus, $p_0$ is referred to as the begin point and $p_3$ as the end point. (Collectively, $p_0$ and $p_3$ are often referred to as end points.) The points $p_1$ and $p_2$ are called control points. The control points function like magnets to pull the curve toward them. Here's a sample Bézier curve showing the two end points and two control points:

Notice how the curve begins at $p_0$ by heading toward $p_1$ but then abandons that trip and heads toward $p_2$. Not touching $p_2$ either, the curve ends at $p_3$. Here's another Bézier curve:
Only rarely does the Bézier curve pass through the two control points. However, if you position both control points between the end points, the Bézier curve becomes a straight line and passes through them:

At the other extreme, it's even possible to choose points that make the Bézier spline do a loop:

To draw a Bézier curve in a Windows Forms program, you need to specify the four points, either as four `Point` or `PointF` structures or as eight `float` values:

**Graphics DrawBezier Methods**

```csharp
void DrawBezier(Pen pen, Point pt0, Point pt1, Point pt2, Point pt3)
void DrawBezier(Pen pen, PointF ptf0, PointF ptf1, PointF ptf2, PointF ptf3)
void DrawBezier(Pen pen, float x0, float y0, float x1, float y1, float x2, float y2, float x3, float y3)
```

It's sometimes more convenient to specify these four points as an array of `Point` or `PointF` structures. The two `DrawBeziers` methods let you do that. (Notice the plural.) You can pass an array of four `Point` or `PointF` structures to the `DrawBeziers` method, or you can use the method to draw multiple connected Bézier splines:

**Graphics DrawBeziers Methods**

```csharp
void DrawBeziers(Pen pen, Point[] apt)
void DrawBeziers(Pen pen, PointF[] aptf)
```

When you draw multiple Bézier splines, the end point of each connected spline is the same as the begin point of the next spline, which means that each additional spline requires three more points. To draw $N$ Bézier curves, the number of points in the array must be equal to $3N + 1$. If the size of the array doesn't equal $3N + 1$, for $N \geq 1$, the method throws an exception.

There are no `FillBezier` or `FillBeziers` methods. If you want to use Bézier curves to fill enclosed areas, you need to use graphics paths, which I cover in Chapter 15.

You can get a good feel for the Bézier curve by experimenting with the following program.

*Bezier.cs*
using System;
using System.Drawing;
using System.Windows.Forms;

class Bezier: Form
{
    protected Point[] apt = new Point[4];

    public static void Main()
    {
        Application.Run(new Bezier());
    }
    public Bezier()
    {
        Text = "Bezier (Mouse Defines Control Points)";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
        OnResize(EventArgs.Empty);
    }
    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);
        int cx = ClientSize.Width;
        int cy = ClientSize.Height;

        apt[0] = new Point(cx / 4, cy / 2);
        apt[1] = new Point(cx / 2, cy / 4);
        apt[2] = new Point(cx / 2, 3 * cy / 4);
        apt[3] = new Point(3 * cx / 4, cy / 2);
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        Point pt;

        if (mea.Button == MouseButtons.Left)
            pt = apt[1];
        else if (mea.Button == MouseButtons.Right)
            pt = apt[2];
    }
else
    return;

    Cursor.Position = PointToScreen(pt);
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    if (mea.Button == MouseButtons.Left)
    {
        apt[1] = new Point(mea.X, mea.Y);
        Invalidate();
    }
    else if (mea.Button == MouseButtons.Right)
    {
        apt[2] = new Point(mea.X, mea.Y);
        Invalidate();
    }
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    grfx.DrawBeziers(new Pen(ForeColor), apt);

    Pen pen = new Pen(Color.FromArgb(0x80, ForeColor));

    grfx.DrawLine(pen, apt[0], apt[1]);
    grfx.DrawLine(pen, apt[2], apt[3]);
}

The program fixes the two end points and lets you manipulate the two control points with the mouse. Use the left mouse button for $p_1$ and the right mouse button for $p_2$. I implemented a "snap to" feature in this program: when you press the left or right mouse button, the program uses the static Cursor.Position property to move the position of the mouse cursor to the appropriate control point. The program also draws gray lines from the end points to the control points. Here's a typical display:
Bézier splines are useful in computer-assisted graphics design work because of several characteristics. First, with a little practice, you can usually manipulate the curve into something close to the shape you desire.

Second, the Bézier spline is very well controlled. Some splines don't pass through any of the points that define them. The Bézier spline is always anchored at the two end points. (As we'll see, this is one of the assumptions that is used to derive the Bézier formulas.) Also, some forms of splines have singularities where the curve veers off into infinity (an effect rarely desired in computer-design work). The Bézier spline is much better behaved. In fact, the Bézier spline is always bounded by a four-sided polygon (called a convex hull) that is formed by connecting the end points and the control points. (The way in which you connect the end points and the control points to form this convex hull depends on the particular curve.)

The third characteristic of the Bézier spline involves the relationship between the end points and the control points. At the begin point, the curve is always tangential to and in the same direction as a straight line drawn from the begin point to the first control point. (This relationship is visually illustrated in the Bézier program.) At the end point, the curve is always tangential to and in the same direction as a straight line drawn from the second control point to the end point. These are two other assumptions used to derive the Bézier formulas.

Fourth, the Bézier spline is often aesthetically pleasing. I know this is a subjective criterion, but I'm not the only person who finds the Bézier curve quite graceful.

**A More Stylish Clock**

In the two decades since the dawn of analog clock programs, such programs have looked pretty much the same. Almost always the programmer uses a fairly simple polygon to draw the hands of the clock. It is now time to explore new vistas by drawing the clock hands using Bézier splines.

You'll recall that the AnalogClock program in Chapter 10, "The Timer and Time," made use of a control that I implemented in a class named ClockControl. Fortunately, I had the foresight to isolate the clock hand-drawing code in protected virtual methods in that class. Here's a BezierClockControl class that makes calls to DrawBeziers in new DrawHourHand and DrawMinuteHand methods.

**BezierClockControl.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
```
using System.Windows.Forms;

namespace Petzold.ProgrammingWindowsWithCSharp
{
    class BezierClockControl: ClockControl
    {
        protected override void DrawHourHand(Graphics grfx, Pen pen)
        {
            GraphicsState gs = grfx.Save();
            grfx.RotateTransform(360f * Time.Hour / 12 +
                30f * Time.Minute / 60);

            grfx.DrawBeziers(pen, new Point[]
            {
                new Point(  0, -600), new Point(  0, -300),
                new Point(200, -300), new Point(  50, -200),
                new Point(50, -200), new Point(  50,  0),
                new Point(50,  0), new Point(  50,  75),
                new Point(-50,  75), new Point( -50,  0),
                new Point(-50,  0), new Point( -50, -200),
                new Point(-50, -200), new Point(-200, -300),
                new Point(  0, -300), new Point(  0, -600)
            });
            grfx.Restore(gs);
        }

        protected override void DrawMinuteHand(Graphics grfx, Pen pen)
        {
            GraphicsState gs = grfx.Save();
            grfx.RotateTransform(360f * Time.Minute / 60 +
                6f * Time.Second / 60);

            grfx.DrawBeziers(pen, new Point[]
            {
                new Point(  0, -800), new Point(  0, -750),
                new Point( 25, -600), new Point( 25, -600),
                new Point(25, -600), new Point( 25,  0),
                new Point(25,  0), new Point( 25,  50),
                new Point(-25,  50), new Point( -25,  0),
                new Point(-25,  0), new Point( -25, -600),
                new Point(-25, -600), new Point(  0, -700),
                new Point(  0, -750), new Point(  0, -800)
            });
            grfx.Restore(gs);
        }
    }
}
Each of the two calls to \textit{DrawBeziers} passes an array of 16 \textit{Point} structures to draw 5 Bézier curves. (Remember that the first Bézier curve drawn by \textit{DrawBeziers} requires 4 points; each subsequent curve requires 3 more.)

The original \textit{AnalogClock} program was so small that I decided it didn't make sense trying to subclass it. Instead, here's a brand new \textit{BezierClock} program that takes advantage of the \textit{BezierClockControl} class.

\textbf{BezierClock.cs}

```
// BezierClock.cs © 2001 by Charles Petzold
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierClock: Form
{
    BezierClockControl clkctl;

    public static void Main()
    {
        Application.Run(new BezierClock());
    }
    public BezierClock()
    {
        Text = "Bezier Clock";
        clkctl = new BezierClockControl();
        clkctl.Parent = this;
        clkctl.Time = DateTime.Now;
        clkctl.Dock = DockStyle.Fill;
        clkctl.BackColor = Color.Black;
        clkctl.ForeColor = Color.White;
        Timer timer = new Timer();
        timer.Interval = 100;
        timer.Tick += new EventHandler(OnTimerTick);
        timer.Start();
    }
    void OnTimerTick(object obj, EventArgs ea)
    {
        clkctl.Time = DateTime.Now;
    }
}
```
And here it is:

The curved tip of each hand is defined by two Bézier curves, one on each side. The straight-line portions are another pair of Bézier curves, and the rounded part at the center of the clock is another curve, for a total of five.

**Collinear Béziers**

Although connected Bézier curves share end points, it's possible that the point at which one curve ends and the other begins won't be smooth. Mathematically speaking, the composite curve is considered to be smooth only if the first derivative of the curve is continuous—that is, it doesn't make any sudden changes.

When you draw multiple Bézier curves, you may want the resultant composite curve to be smooth where one curve ends and the next one begins. Then again, you may not. The hands of the clock have a combination of smoothness and discontinuity. The point at which the two Bézier curves meet at the tip of the clock hand has a discontinuous first derivative. Likewise, there's a discontinuity where the Bézier curve defining the curved part of the tip meets the straight line. However, the straight lines smoothly join the rounded part at the center of the clock.

If you want connected Bézier curves to join each other smoothly, the following three points must be collinear (that is, lie on the same line):
- The second control point of the first Bézier
- The end point of the first Bézier (which is the same as the begin point of the second Bézier)
- The first control point of the second Bézier

Here's a program that draws four connected Bézier curves that are smooth at each connection. The end of the fourth Bézier curve meets the beginning of the first curve to create a closed curve.

```csharp
Infinity.cs

using System;
using System.Drawing;
using System.Windows.Forms;

class Infinity: PrintableForm
{
    public new static void Main()
```
public Infinity()
{
    Text = "Infinity Sign Using Bezier Splines";
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    cx--;
    cy--;

    Point[] apt =
    {
        new Point(0, cy / 2),  // Begin
        new Point(0, 0),      // Control
        new Point(    cx / 3, 0),   // Control
        new Point(    cx / 2, cy / 2),   // End / Begin
        new Point(2 * cx / 3, cy),  // Control
        new Point(    cx, cy),  // Control
        new Point(    cx, cy / 2),  // End / Begin
        new Point(    cx, 0),   // Control
        new Point(2 * cx / 3, 0),  // Control
        new Point(    cx / 2, cy / 2),   // End / Begin
        new Point(    cx / 3, cy),   // Control
        new Point(0, cy),         // Control
        new Point(0, cy / 2)      // End
    };
    grfx.DrawBeziers(new Pen(clr), apt);
}

In the array, each point labeled Begin, End, or End/Begin is collinear with the two control points on either side. The result of these four Bézier splines is a design that somewhat resembles an infinity sign:
Circles and Arcs with Béziers

Earlier in this chapter, I mentioned that PostScript uses Bézier splines to draw elliptical arcs. As you’ll discover in Chapter 15, Windows Forms does so as well, at least when it comes time to store arcs and ellipses to a graphics path.

A couple articles that describe the approximation of elliptical arcs using Bézier splines are available. The first of these articles describes a fairly simple technique that you can use to draw segments of a circle. Suppose you want to use a Bézier spline to draw a circular arc with a particular radius and an angular width of $\alpha$. You know that you must set $p_0$ and $p_3$ to the points at the beginning and the end of the arc, but how do you set $p_1$ and $p_2$? As this diagram illustrates, the problem reduces to finding the distance between the end points and control points—a single length labeled $L$:

I’ve indicated that the lines connecting the end points with the control points are at right angles to the radii. How do we know this? Because of the collinearity requirement for smoothness. If you were to use a Bézier spline to draw another arc adjacent to this one with the same center and radius, the common end point and the two adjacent control points would need to be collinear. That means that the line from the end point to the control point is at right angles to the circle’s radius.

If you know $L$, calculating the coordinates of $p_1$ and $p_2$ involves just basic trigonometry. But look how simple the calculations of $p_1$ and $p_2$ are when you use an angle of 90 degrees oriented with the horizontal and vertical coordinates:
The calculation of $p_1$ and $p_2$ is also trivial when you use an angle of 180 degrees.

The first paper I cited demonstrates that a fairly good approximation results from

$$L = \frac{4}{3} \tan \left( \frac{1}{4} \alpha \right)$$

times the radius.

The BezierCircles program draws two complete circles using this approximation, first using two Bézier splines and then (more accurately) using four Bézier splines.

```csharp
beziers.cs
//------------------------------------------------------------------------------
// beziers.cs © 2001 by Charles Petzold
//------------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierCircles : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new BezierCircles());
    }
    public BezierCircles()
    {
        Text = "Bezier Circles";
    }
}```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    int iRadius = Math.Min(cx - 1, cy - 1) / 2;

    grfx.DrawEllipse(new Pen(clr), cx / 2 - iRadius, cy / 2 - iRadius, 2 * iRadius, 2 * iRadius);

    // Two-segment (180-degree) approximation
    int L = (int)Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 4));
    Point[] apt = {
        new Point(cx / 2, cy / 2 - iRadius),
        new Point(cx / 2 + L, cy / 2 - iRadius),
        new Point(cx / 2 + L, cy / 2 + iRadius),
        new Point(cx / 2, cy / 2 + iRadius),
        new Point(cx / 2 - L, cy / 2 + iRadius),
        new Point(cx / 2 - L, cy / 2 - iRadius),
        new Point(cx / 2, cy / 2 - iRadius)
    };
    grfx.DrawBeziers(Pens.Blue, apt);

    // Four-segment (90-degree) approximation
    L = (int)Math.Round(iRadius * 4f / 3 * Math.Tan(Math.PI / 8));
    apt = new Point[]
    {
        new Point(cx / 2, cy / 2 - iRadius),
        new Point(cx / 2 + L, cy / 2 - iRadius),
        new Point(cx / 2 + iRadius, cy / 2 - L),
        new Point(cx / 2 + iRadius, cy / 2),
        new Point(cx / 2 + iRadius, cy / 2 + L),
        new Point(cx / 2 + L, cy / 2),
        new Point(cx / 2, cy / 2 + L),
        new Point(cx / 2 - L, cy / 2 + L),
        new Point(cx / 2 - iRadius, cy / 2 + L),
        new Point(cx / 2 - iRadius, cy / 2),
        new Point(cx / 2 - iRadius, cy / 2 - L),
    };
    grfx.DrawBeziers(Pens.Blue, apt);
This program also visually demonstrates how the Bézier approximation differs from the `DrawEllipse` method. The program begins its `DoPage` processing by calling `DrawEllipse` to draw an ellipse in black. The two-Bézier approximation is drawn in blue, and the four-Bézier version in red. Remember that the arguments to the `Math` class trigonometric functions are in units of radians, so instead of dividing the angle by 4 as the formula for $L$ indicates, I use an expression based on the `Math.PI` constant.


**Bézier Art**

Many people—including Pierre Bézier himself—have used Bézier splines to create interesting designs and patterns. These are generally lumped under the category of "Bézier art." There are no rules here except that a for loop is generally involved. Here's an example.

**BezierArt.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierArt: PrintableForm {
    const int iNum = 100;

    public new static void Main()
    {
        Application.Run(new BezierArt());
    }

    public BezierArt()
    {
        Text = "Bezier Art";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr);
        PointF[] aptf = new PointF[4];

        for (int i = 0; i < iNum; i++)
        {
```
Images that involve a lot of line or curve drawing usually look better when printed, but here's the video version of this one:

Although I'm stuck with black and white for images in this book, don't forget about color when you do your own Bézier art.


The Mathematical Derivation

It's sometimes helpful to know the underlying formulas that a graphics system uses to render particular curves. For example, you may need to orient other graphics figures (text characters, perhaps) in relationship to a curve that the system has drawn. It's also a good exercise to derive the curves, if only so that you don't think the formulas fell out of the sky one day.
A Bézier spline is a cubic polynomial. Like all cubic polynomials, a Bézier spline is uniquely defined by four points, which we have called \( p_0 \) (the begin point), \( p_1 \) and \( p_2 \) (the two control points), and \( p_3 \) (the end point). These four points can also be denoted as \((x_0, y_0), (x_1, y_1), (x_2, y_2), \) and \((x_3, y_3)\).

The general parametric form of a cubic polynomial in two dimensions is

\[
x(t) = ax \cdot t^3 + bx \cdot t^2 + cx \cdot t + dx \\
y(t) = ay \cdot t^3 + by \cdot t^2 + cy \cdot t + dy
\]

where \(ax, bx, cx, dx, ay, by, cy, \) and \(dy\) are constants, and \(t\) ranges from 0 to 1. Every Bézier spline is uniquely defined by these eight constants. The constants are dependent on the four points that define the spline. The object of this exercise is to develop equations for the eight constants in terms of the four points.

The first assumption is that the Bézier spline begins at the point \((x_0, y_0)\) when \(t\) equals 0:

\[
x(0) = x_0 \\
y(0) = y_0
\]

Even with this simple assumption we can make some headway in deriving the constants. If you put a 0 value for \(t\) in the parametric equations, you get

\[
x(0) = dx \\
y(0) = dy
\]

This means that two of the constants are simply the coordinates of the begin point:

\[
d_x = x_0 \quad (1a) \\
d_y = y_0 \quad (1b)
\]

The second assumption regarding the Bézier spline is that it ends at the point \((x_3, y_3)\) when \(t\) equals 1:

\[
x(1) = x_3 \\
y(1) = y_3
\]

Substituting a value of 1 for \(t\) in the parametric formulas yields the following:

\[
x(1) = ax + bx + cx + dx \\
y(1) = ay + by + cy + dy
\]

This means that the constants relate to the coordinate of the end point like so:

\[
a_x + b_x + c_x + d_x = x_3 \quad (2a) \\
a_y + b_y + c_y + d_y = y_3 \quad (2b)
\]

The remaining two assumptions involve the first derivatives of the parametric equations, which describe the slope of the curve. The first derivatives of the generalized parametric equations of a cubic polynomial with respect to \(t\) are

\[
x'(t) = 3ax t^2 + 2bx t + cx \\
y'(t) = 3ay t^2 + 2by t + cy
\]

In particular, we're interested in the slope of the curve at the two end points. At the begin point, the Bézier spline is tangential to and in the same direction as a straight line drawn from the first begin point to the first control point. That straight line would normally be defined by the parametric equations

\[
x(t) = (x_1 - x_0) \cdot t + x_0 \\
y(t) = (y_1 - y_0) \cdot t + y_0
\]

for \(t\) ranging from 0 to 1. However, another way of expressing this straight line would be the parametric equations

\[
x(t) = (x_1 - x_0) \cdot t + x_0 \\
y(t) = (y_1 - y_0) \cdot t + y_0
\]
\[x(t) = 3\ (x_1 - x_0)\ t + x_0\]
\[y(t) = 3\ (y_1 - y_0)\ t + y_0\]

where \(t\) ranges from 0 to 1/3. Why 1/3? Because the section of the Bézier spline that is tangential to and in the same direction as the straight line from \(p_2\) to \(p_3\) is roughly 1/3 of the total Bézier spline.

Here are the first derivatives of these revised parametric equations:

\[x'(t) = 3\ (x_1 - x_0)\]
\[y'(t) = 3\ (y_1 - y_0)\]

We want these equations to represent the slope of the Bézier spline when \(t\) equals 0, so

\[x'(0) = 3\ (x_1 - x_0)\]
\[y'(0) = 3\ (y_1 - y_0)\]

Substitute \(t\) in the generalized cubic first derivatives, and you get

\[x'(0) = c_x\]
\[y'(0) = c_y\]

That means

\[c_x = 3\ (x_1 - x_0)\]  \hspace{1cm} (3a)
\[c_y = 3\ (y_1 - y_0)\]  \hspace{1cm} (3b)

The last assumption is that at the end point, the Bézier spline is tangential to and in the same direction as a straight line drawn from the second control point to the end point. In other words,

\[x'(1) = 3\ (x_0 - x_3)\]
\[y'(1) = 3\ (y_0 - y_3)\]

Since we know from the generalized formulas that

\[x'(1) = 3ax + 2bx + cx\]
\[y'(1) = 3ay + 2by + cy\]

then

\[3a_x + 2b_x + c_x = 3\ (x_0 - x_3)\]  \hspace{1cm} (4a)
\[3a_y + 2b_y + c_y = 3\ (y_0 - y_3)\]  \hspace{1cm} (4b)

Equations 1a, 2a, 3a, and 4a provide four equations and four unknowns that let us solve for \(a_x\), \(b_x\), \(c_x\), and \(d_x\) in terms of \(x_0\), \(x_1\), \(x_2\), and \(x_3\). Go through the algebra, and you find

\[a_x = -x_0 + 3x_1 - 3x_2 + x_3\]
\[b_x = 3x_0 - 6x_1 + 3x_2\]
\[c_x = 3x_0 + 3x_1\]
\[d_x = x_0\]

Equations 1b, 2b, 3b, and 4b let us do the same for the \(y\) coefficients. We can then put the constants back into the generalized cubic parametric equations:

\[x(t) = (-x_0 + 3x_1 - 3x_2 + x_3)\ t^3 + (3x_0 - 6x_1 + 3x_2)\ t^2 + (3x_0 + 3x_1)\ t + x_0\]
\[y(t) = (-y_0 + 3y_1 - 3y_2 + y_3)\ t^3 + (3y_0 - 6y_1 + 3y_2)\ t^2 + (3y_0 + 3y_1)\ t + y_0\]

We're basically done. However, it's much more common for the terms to be rearranged to yield the more elegant and easier-to-use parametric equations:

\[x(t) = (1 - \theta)^3 \ x_0 + 3\theta (1 - \theta)^2 \ x_1 + 3\theta^2 (1 - \theta) \ x_2 + \theta^3 \ x_3\]
\[y(t) = (1 - \theta)^3 \ y_0 + 3\theta (1 - \theta)^2 \ y_1 + 3\theta^2 (1 - \theta) \ y_2 + \theta^3 \ y_3\]

These equations are the customary form in which the Bézier spline is expressed.
The `BezierManual` class in the following program overrides the `Bezier` class from the Bezier program earlier in this chapter and draws a second Bézier spline—this time calculated “manually” using the parametric equations I just derived.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class BezierManual : Bezier
{
    public new static void Main()
    {
        Application.Run(new BezierManual());
    }
    public BezierManual()
    {
        Text = "Bezier Curve \"Manually\" Drawn";
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        base.OnPaint(pea);

        BezierSpline(pea.Graphics, Pens.Red, apt);
    }

    void BezierSpline(Graphics grfx, Pen pen, Point[] aptDefine)
    {
        Point[] apt = new Point[100];

        for (int i = 0; i < apt.Length; i++)
        {
            float t = (float) i / (apt.Length - 1);

            float x = (1 - t) * (1 - t) * (1 - t) * aptDefine[0].X +
                      3 * t * (1 - t) * (1 - t) * aptDefine[1].X +
                      3 * t * t * (1 - t) * aptDefine[2].X +
                      t * t * t * aptDefine[3].X;

            float y = (1 - t) * (1 - t) * (1 - t) * aptDefine[0].Y +
                      3 * t * (1 - t) * (1 - t) * aptDefine[1].Y +
                      3 * t * t * (1 - t) * aptDefine[2].Y +
                      t * t * t * aptDefine[3].Y;
        }
    }
}
```
apt[i] = new Point((int) Math.Round(x), (int) Math.Round(y));
}
grfx.DrawLines(pen, apt);
}

The OnPaint method in BezierManual calls the OnPaint method in the base class (that's theBezier class) and then calls the BezierSpline method in its own class. The BezierSpline method is defined much the same way as DrawBeziers except that it has a Graphics object as a first argument and is equipped to handle only a single Bézier spline. This method uses an array of 100 Point structures, calculates each Point based on the parametric equations I derived above, and then draws the spline as a polyline. The program draws the manually calculated Bézier spline in red, so you can compare it with the version that Windows Forms draws. It's not exact, but it never differs by more than 1 pixel.

The Canonical Spline

The Graphics class includes a second type of spline called the canonical spline, meaning a standard or normal spline. You draw a canonical spline by using one of the DrawCurve methods. DrawCurve comes in seven different versions, but you'll probably use the following four methods most frequently:

**Graphics DrawCurve Methods (selection)**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawCurve(Pen pen, Point[] apt)</td>
</tr>
<tr>
<td>DrawCurve(Pen pen, PointF[] aptf)</td>
</tr>
<tr>
<td>DrawCurve(Pen pen, Point[] apt, float fTension)</td>
</tr>
<tr>
<td>DrawCurve(Pen pen, PointF[] aptf, float fTension)</td>
</tr>
</tbody>
</table>

At least two points are required. If the array contains only two points, the DrawCurve method draws a straight line from the first point to the second. For three points or more, the method draws a curved line that connects all the points.

The big difference between the Bézier spline and the canonical spline is that the canonical spline passes through every point in the array. The curve between each adjacent pair of points is sometimes called a segment of the total curve. The shape of each segment of the curve is governed by the two points at the beginning and the end of the segment (of course) but also the other two adjacent points. For example, for an array of Point structures named apt, the shape of the segment between apt[3] and apt[4] is also affected by the points apt[2] and apt[5].

The spline is also affected by the tension, which is an explicit argument in some of the DrawCurve overloads. If you think of traditional wooden or metal splines, the tension is equivalent to the stiffness of the spline. The default is 0.5. A tension of 0 results in straight lines: DrawCurve becomes DrawLines. With tensions greater than 0.5, the curve gets curvier. You can set tensions less than 0, but they often result in loops. Tensions much higher than 1 can also create loops.

Let's experiment. The following program is much like the Bezier program except that it also includes a scroll bar for setting the tension and it gives you more flexibility in moving the points around.

**CanonicalSpline.cs**

```csharp
//----------------------------------------------
// CanonicalSpline.cs © 2001 by Charles Petzold
//----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
```
class CanonicalSpline: Form
{
    protected Point[] apt = new Point[4];
    protected float fTension = 0.5f;

    public static void Main()
    {
        Application.Run(new CanonicalSpline());
    }

    public CanonicalSpline()
    {
        Text = "Canonical Spline";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;

        ScrollBar scroll = new VScrollBar();
        scroll.Parent = this;
        scroll.Dock = DockStyle.Right;
        scroll.Minimum = -100;
        scroll.Maximum = 109;
        scroll.SmallChange = 1;
        scroll.LargeChange = 10;
        scroll.Value = (int)(10 * fTension);
        scroll.ValueChanged += new EventHandler(ScrollOnValueChanged);

        OnResize(EventArgs.Empty);
    }

    void ScrollOnValueChanged(object obj, EventArgs ea)
    {
        ScrollBar scroll = (ScrollBar)obj;

        fTension = scroll.Value / 10f;

        Invalidate(false);
    }

    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);

        int cx = ClientSize.Width;
        int cy = ClientSize.Height;

        apt[0] = new Point(cx / 4, cy / 2);
        apt[1] = new Point(cx / 2, cy / 4);
    }
}
apt[2] = new Point( cx / 2, 3 * cy / 4);
apt[3] = new Point(3 * cx / 4, cy / 2);
}

protected override void OnMouseDown(MouseEventArgs mea)
{
    Point pt;

    if (mea.Button == MouseButtons.Left)
    {
        if (ModifierKeys == Keys.Shift)
            pt = apt[0];
        else if (ModifierKeys == Keys.None)
            pt = apt[1];
        else
            return;
    }
    else if (mea.Button == MouseButtons.Right)
    {
        if (ModifierKeys == Keys.None)
            pt = apt[2];
        else if (ModifierKeys == Keys.Shift)
            pt = apt[3];
        else
            return;
    }
    else
        return;

    Cursor.Position = PointToScreen(pt);
}

protected override void OnMouseMove(MouseEventArgs mea)
{
    Point pt = new Point(mea.X, mea.Y);

    if (mea.Button == MouseButtons.Left)
    {
        if (ModifierKeys == Keys.Shift)
            apt[0] = pt;
        else if (ModifierKeys == Keys.None)
            apt[1] = pt;
        else
            return;
    }
    else if (mea.Button == MouseButtons.Right)
    {
if (ModifierKeys == Keys.None)
    apt[2] = pt;
else if (ModifierKeys == Keys.Shift)
    apt[3] = pt;
else
    return;
}
else
    return;

Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Brush       brush = new SolidBrush(ForeColor);

    grfx.DrawCurve(new Pen(ForeColor), apt, fTension);

    grfx.DrawString("Tension = " + fTension, Font, brush, 0, 0);

    for (int i = 0; i < 4; i++)
        grfx.FillEllipse(brush, apt[i].X - 3, apt[i].Y - 3, 7, 7);
}

As with the Bezier program, you use the left mouse button and the right mouse button to change the locations of $p_1$ and $p_2$. In addition, the CanonicalSpline program lets you change the locations of $p_0$ and $p_3$ by using the left and right mouse buttons in conjunction with the Shift key. Here's a typical display:

![Typical display of the Canonical Spline program](image)

You adjust the tension with the scroll bar; the value is displayed in the upper left corner of the window. I've allowed the tension to range between −10 and 10, just so you can see for yourself how
extreme values make the curve go crazy. Here's one of my favorites using the program's default setting of the Point array:

![Canonical Spline](image)

It's also possible to use a subset of the point array in the following DrawCurve methods:

**Graphics DrawCurve Methods (selection)**

```csharp
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments)
DrawCurve(Pen pen, PointF[] aptf, int iOffset, int iSegments, float fTension)
```

Think of the iOffset argument as an index into the Point or PointF array. That's where the curve begins. The iSegments argument indicates the number of segments drawn and also the number of additional Point or PointF structures the method will use. For example, suppose aptf is an array of PointF structures. The call
gfx.DrawCurve(pen, aptf, 2, 3);

draws three segments, from aptf[2] to aptf[3], from aptf[3] to aptf[4], and from aptf[4] to aptf[5]. The visual results aren't the same as calling the simpler version of DrawCurve with just these four points. The versions with iOffset and iSegments use the aptf[1] point in determining the shape of the curve from aptf[2] to aptf[3], and the aptf[6] point for the curve between aptf[4] and aptf[5].

The DrawClosedCurve methods connect the last point in the array to the first point in the array with an additional curve:

**Graphics DrawClosedCurve Methods**

```csharp
DrawClosedCurve(Pen pen, Point[] apt)
DrawClosedCurve(Pen pen, PointF[] aptf)
DrawClosedCurve(Pen pen, Point[] apt, float fTension, FillMode fm)
DrawClosedCurve(Pen pen, PointF[] aptf, float fTension, FillMode fm)
```
DrawClosedCurve does more than simply draw an additional segment. The first segment drawn by DrawClosedCurve is a little different than the segment drawn by DrawCurve because it is influenced by the last point in the array; similarly, the penultimate curve is influenced by the first point in the array.

Two of the DrawClosedCurve overloads have a FillMode argument. Of course you remember FillMode, an enumeration defined in the namespace System.Drawing.Drawing2D that is used in the DrawPolygon method to govern which enclosed areas are filled:

<table>
<thead>
<tr>
<th>FillMode Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Alternate</td>
</tr>
<tr>
<td>Winding</td>
</tr>
</tbody>
</table>

But why—you ask—is a fill mode required in a method that simply draws lines and doesn't fill? It's a mystery, and the methods seem to work the same regardless of the FillMode setting.

The FillMode argument makes a lot more sense in the FillClosedCurve methods:

<table>
<thead>
<tr>
<th>Graphics FillClosedCurve Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>FillClosedCurve(Brush brush, Point[] apt)</td>
</tr>
<tr>
<td>FillClosedCurve(Brush brush, PointF[] aptf)</td>
</tr>
<tr>
<td>FillClosedCurve(Brush brush, Point[] apt, FillMode fm)</td>
</tr>
<tr>
<td>FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm)</td>
</tr>
<tr>
<td>FillClosedCurve(Brush brush, Point[] apt, FillMode fm, float fTension)</td>
</tr>
<tr>
<td>FillClosedCurve(Brush brush, PointF[] aptf, FillMode fm, float fTension)</td>
</tr>
</tbody>
</table>

The ClosedCurveFillModes program shown next is almost identical to the FillModesClassical program from Chapter 5. The program draws two five-pointed stars to illustrate the difference between FillMode.Alternate and FillMode.Winding.

ClosedCurveFillModes.cs

```csharp
//---------------------------------------------------
// ClosedCurveFillModes.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClosedCurveFillModes: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new ClosedCurveFillModes());
    }
    ClosedCurveFillModes()
    {
        Text = "FillClosedCurve Fill Modes";
    }
}```
While still recognizable as stars, these figures have a softer look:

They look more like star-shaped cookies that came out of the cookie cutter with straight sides but then plumped up a little when baking.

**Canonical Curve Derivation**

Like the Bézier spline, the canonical spline is a cubic, so it has the general parametric formulas

\[
\begin{align*}
x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\
y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y
\end{align*}
\]

for \( t \) ranging from 0 to 1. The first derivatives are

\[
\begin{align*}
x'(t) &= 3a_x t^2 + 2b_x t + c_x \\
y'(t) &= 3a_y t^2 + 2b_y t + c_y
\end{align*}
\]
Let's look at four points, labeled $p_0$, $p_1$, $p_2$, and $p_3$. I'm going to develop the formulas for the segment between $p_1$ and $p_2$. That curve is based on those two points as well as the two adjoining points, $p_0$ and $p_3$. The first assumptions are that the curve begins at $p_1$ and ends at $p_2$:

$$
\begin{align*}
x(0) &= x_1 \\
y(0) &= y_1 \\
x(1) &= x_2 \\
y(1) &= y_2
\end{align*}
$$

From the generalized parametric formulas, we can then derive the equations

$$
\begin{align*}
d_x &= x_1 \\
d_y &= y_1 \\
a_x + b_x + c_x + d_x &= x_2 \\
a_y + b_y + c_y + d_y &= y_2
\end{align*}
$$

The other two assumptions govern the slope of the line at $p_1$ and $p_2$. The slope at $p_1$ is assumed to be the product of the tension (which I'll represent as $T$) and the slope of the straight line between $p_0$ and $p_2$. Similarly, the slope at $p_2$ is assumed to be the tension times the straight-line slope between $p_1$ and $p_3$:

$$
\begin{align*}
x'(0) &= T(x_2 - x_0) \\
y'(0) &= T(y_2 - y_0) \\
x'(1) &= T(x_3 - x_1) \\
y'(1) &= T(y_3 - y_1)
\end{align*}
$$

From the first derivatives of the general parametric formulas, we find that

$$
\begin{align*}
c_x &= T(x_3 - x_0) \\
c_y &= T(y_3 - y_0) \\
3a_x + 2b_x + c_x &= T(x_3 - x_1) \\
3a_y + 2b_y + c_y &= T(y_3 - y_1)
\end{align*}
$$

With a bit of algebra, solving the simultaneous equations yields

$$
\begin{align*}
a_x &= T(x_0 - x_0) + T(x_3 - x_1) + 2x_1 - 2x_0 \\
a_y &= T(y_0 - y_0) + T(y_3 - y_1) + 2y_1 - 2y_0 \\
b_x &= -2T(x_0 - x_0) - T(x_0 - x_1) - 3x_1 + 3x_0 \\
b_y &= -2T(y_0 - y_0) - T(y_0 - y_1) - 3y_1 + 3y_0 \\
c_x &= T(x_2 - x_0) \\
c_y &= T(y_2 - y_0) \\
d_x &= x_1 \\
d_y &= y_1
\end{align*}
$$

The CanonicalSplineManual program demonstrates that these constants are correct.

```csharp
CanonicalSplineManual.cs
//----------------------------------------------------
// CanonicalSplineManual.cs © 2001 by Charles Petzold
//----------------------------------------------------

using System;
using System.Drawing;
using System.Windows.Forms;

class CanonicalSplineManual: CanonicalSpline
{
    public new static void Main()
    {
        Application.Run(new CanonicalSplineManual());
    }
}
```
public CanonicalSplineManual()
{
    Text = "Canonical Spline \"Manually\" Drawn";
}
protected override void OnPaint(PaintEventArgs pea)
{
    base.OnPaint(pea);
    CanonicalSpline(pea.Graphics, Pens.Red, apt, fTension);
}
void CanonicalSpline(Graphics grfx, Pen pen, Point[] apt, float T)
{
    CanonicalSegment(grfx, pen, apt[0], apt[0], apt[1], apt[2], T);
    CanonicalSegment(grfx, pen, apt[0], apt[1], apt[2], apt[3], T);
    CanonicalSegment(grfx, pen, apt[1], apt[2], apt[3], apt[3], T);
}
void CanonicalSegment(Graphics grfx, Pen pen, Point pt0, Point pt1, 
                       Point pt2, Point pt3, float T)
{
    Point[] apt = new Point[10];
    float SX1 = T * (pt2.X - pt0.X);
    float SY1 = T * (pt2.Y - pt0.Y);
    float SX2 = T * (pt3.X - pt1.X);
    float SY2 = T * (pt3.Y - pt1.Y);
    float AX = SX1 + SX2 + 2 * pt1.X - 2 * pt2.X;
    float AY = SY1 + SY2 + 2 * pt1.Y - 2 * pt2.Y;
    float BX = -2 * SX1 - SX2 - 3 * pt1.X + 3 * pt2.X;
    float BY = -2 * SY1 - SY2 - 3 * pt1.Y + 3 * pt2.Y;
    float CX = SX1;
    float CY = SY1;
    float DX = pt1.X;
    float DY = pt1.Y;
    for (int i = 0; i < apt.Length; i++)
    {
        float t = (float)i / (apt.Length - 1);
        apt[i].X = (int) (AX * t * t * t + BX * t * t + CX * t + 
                        DX);
        apt[i].Y = (int) (AY * t * t * t + BY * t * t + CY * t + 
                        DY);
    }
    grfx.DrawLines(pen, apt);
}
I want to point out a couple things here. The CanonicalSpline method only handles a four-element array and calls CanonicalSegment three times, each time displaying one of the three segments. The first segment and the last segment require special treatment because the curve is based on only three points rather than four.

The CanonicalSegment method uses an array of only 10 Point structures for each of the segments. That's not quite enough for a smooth curve, but it's enough to demonstrate that the method does indeed mimic the DrawCurve method implemented in the Graphics class.

I'll have some more sample programs using Bézier splines and canonical splines in Chapters 15 and 19.
Chapter 14: Menus

Overview

The menu is the focal point of most traditional Microsoft Windows applications. Residing just under the form's title bar, the menu essentially lists everything that the program is capable of doing—from simple operations like cut and paste to complex jobs like spelling checks. Even if an application supports a large number of function-key shortcuts, these shortcuts generally duplicate menu items.

The menus of many Windows applications look roughly similar. This consistency is an important aspect of the Windows user interface. Users learn a new program more quickly if the menu works like the menus in other Windows programs. When designing your programs' menus, you should look at existing Windows applications for hints about structure and content. This is not necessarily to say that you should perpetuate any inelegant design choices, but sometimes even an imperfect user interface can be good merely because it's consistent with other applications.

Visual Studio .NET includes a Menu Designer that lets you interactively piece together the hierarchy of your program's menu. The Menu Designer is fairly easy to use, and it's adequate for creating simple menus. However, I can almost guarantee that someday soon you'll need to go beyond the capabilities of this Menu Designer, and you won't much like the code that it generates. For this reason, I'm going to approach menu design in this chapter strictly with code.

Menus and Menu Items

The menu that sits between a form's title bar and the client area is referred to in Windows Forms as the form's main menu. Many applications also support shortcut menus, or context menus, which are menus that appear at the mouse cursor position when you right-click the mouse. A main menu is associated with a form, while a context menu is usually associated with a particular control; that is, clicking different controls often causes different context menus to be invoked.

A menu—either a main menu or a context menu—contains menu items. A menu item is generally associated with a word or a short phrase, such as File, Open, Save, Save As, Edit, Cut, Copy, Paste, or Select All. These are all menu items. I'll often refer to a menu item simply by the text associated with that item.

As you're undoubtedly aware, however, the File and Edit items seem quite different from Open, Save, Save As, Cut, Copy, and Select All. The File and Edit items are located on the visible part of the program's main menu; the others I mentioned are not. The items that run across the visible length of the main menu are known as top-level items. Selecting File or another top-level item from the main menu invokes the display of a rectangular area traditionally called a pop-up menu or a drop-down menu, but nowadays more commonly called a submenu or a child menu. (Yes, here's another parent-children relationship in Windows Forms!) The submenu invoked from the File item contains the additional menu items Open, Save, Save As, and so forth.

From the perspective of your Windows Forms program, the File menu item contains an array of other menu items; this array includes Open, Save, and Save As. The Edit menu item contains an array of menu items that includes Cut, Copy, and Select All.

In fact, if we step backward a moment, we can see that the main menu itself is an array of menu items—an array including File and Edit and everything else in the visible part of the main menu, probably ending with Help. Each menu item in the main menu is associated with its own array of menu items; each of these arrays represents a submenu of the main menu. Some of the menu items in these submenus also include their own arrays of menu items to invoke further nested submenus. Similarly, a context menu is an array of menu items, each of which can include additional arrays of menu items.

The MainMenu, ContextMenu, and MenuItem classes are all derived from the abstract Menu class in the System.Windows.Forms namespace. Here's the class hierarchy:
The first thing you should notice is that Menu isn't derived from Control, so such familiar properties as BackColor, ForeColor, and Font aren't available. Users can change the color and font of menus, but programs can't. If you want to display menus in nonstandard colors and fonts, you'll have to use the owner-draw facility I describe toward the end of this chapter.

I said that a form's main menu is an array of menu items. Here are the two constructors for MainMenu, the second of which clearly indicates this fact:

**MainMenu Constructors**

MainMenu()

MainMenu(MenuItem[] ami)

To attach a particular MainMenu object to a form, you assign it to the form's Menu property:

<table>
<thead>
<tr>
<th>Form Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>MainMenu</td>
</tr>
</tbody>
</table>

Similarly, ContextMenu also has two constructors, the second of which also indicates that a context menu is an array of menu items:

**ContextMenu Constructors**

ContextMenu()

ContextMenu(MenuItem[] ami)

The ContextMenu property of Control lets you attach a particular ContextMenu object to any control:

<table>
<thead>
<tr>
<th>Control Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>ContextMenu</td>
</tr>
</tbody>
</table>

If you have a bunch of controls—or if you've divided your form into different functional areas using panels—each control can invoke a different context menu. However, it's not necessary to use the ContextMenu property of the control to implement context menus. You can instead "manually" invoke different context menus during the OnMouseUp method based on the current mouse cursor position.

Although the constructors for MainMenu and ContextMenu indicate that these objects are associated with arrays of menu items, I haven't shown you the constructors for MenuItem yet. Here are five of the six MenuItem constructors:
**MenuItem Constructors (selection)**

- `MenuItem()`
- `MenuItem(string strText)`
- `MenuItem(string strText, EventHandler ehClick)`
- `MenuItem(string strText, EventHandler ehClick, Shortcut sc)`
- `MenuItem(string strText, MenuItem[] ami)`

`MenuItem` has one additional—and quite complex—constructor that merges menus.

We'll be looking at the `MenuItem` class in much detail throughout this chapter, but it won't hurt to see a couple quick examples now. For a top-level menu item like File, you might use the last constructor in the table:

```csharp
new MenuItem("&File", amiFile)
```

Notice the ampersand that appears before the F in File. The ampersand indicates that the F is to be underlined and that it will be part of the built-in keyboard interface that Windows provides for menus. When the user presses the Alt key and then F, the submenu for File is displayed. Use two ampersands in a row if you want to display an ampersand in the menu text.

The `amiFile` argument to the constructor I've just shown would be an array of menu items for Open, Save, Save As, and so forth. One of the elements of this array is the menu item for Open, which could have been created using the constructor:

```csharp
new MenuItem("&Open...",
    new EventHandler(MenuFileOpenOnClick),
    Shortcut.CtrlO)
```

Again, the ampersand indicates that the O is to be underlined. Pressing Alt, F, and then O causes the Open item to be invoked. The ellipsis indicates to the user that the menu item invokes a dialog box.

Every menu item that does not invoke a submenu is usually associated with a `Click` event handler that is called when the user clicks the item with the mouse or triggers it with the keyboard. I'll be using a standard naming scheme for such event handlers. In a real-life program, the `MenuFileOpenOnClick` method would be responsible for displaying the dialog box that lets the user select a file to open. (I'll be discussing dialog boxes in depth in Chapter 16.)

**Menu Shortcut Keys**

I've mentioned the menu keyboard interface involving underlined letters. When the user presses the Alt key, the form goes into a special menu-selection mode. Pressing the F key displays the File submenu, and pressing O is equivalent to clicking the Open item.

Windows has an additional keyboard interface to the menu, traditionally known as accelerators but in Windows Forms known as shortcuts. You specify a shortcut by using values of the `Shortcut` enumeration. In the previous example, the value `Shortcut.CtrlO` indicates that the Ctrl+O key combination is a shortcut to display a dialog box to open a file. When you use shortcuts with menu items, the text "Ctrl+O" is also automatically inserted in the text of the menu item that is displayed to the user. When the user presses Ctrl+O, the `MenuFileOpenOnClick` event handler is called directly, seemingly without the menu being involved.

`Shortcut` is an enumeration of 150 key combinations recommended for use in menus. (It’s the fourth largest enumeration in the .NET Framework.) The set shown in this first table involves the function keys, Insert, Delete, and Backspace:

<p>| Shortcut Enumeration (selection) |
| --- | --- | --- | --- | --- |
| F1 | ShiftF1 | CtrlF1 | CtrlShiftF1 | AltF1 |
| F2 | ShiftF2 | CtrlF2 | CtrlShiftF2 | AltF2 |</p>
<table>
<thead>
<tr>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3</td>
</tr>
<tr>
<td>F4</td>
</tr>
<tr>
<td>F5</td>
</tr>
<tr>
<td>F6</td>
</tr>
<tr>
<td>F7</td>
</tr>
<tr>
<td>F8</td>
</tr>
<tr>
<td>F9</td>
</tr>
<tr>
<td>F10</td>
</tr>
<tr>
<td>F11</td>
</tr>
<tr>
<td>F12</td>
</tr>
<tr>
<td>Ins</td>
</tr>
<tr>
<td>Del</td>
</tr>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

The default is `Shortcut.None`. The following table shows all the letters in combination with the Shift and Ctrl keys:

### Shortcut Enumeration (selection)

<table>
<thead>
<tr>
<th>CtrlA</th>
<th>CtrlN</th>
<th>CtrlShiftA</th>
<th>CtrlShiftN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CtrlB</td>
<td>CtrlO</td>
<td>CtrlShiftB</td>
<td>CtrlShiftO</td>
</tr>
<tr>
<td>CtrlC</td>
<td>CtrlP</td>
<td>CtrlShiftC</td>
<td>CtrlShiftP</td>
</tr>
<tr>
<td>CtrlD</td>
<td>CtrlQ</td>
<td>CtrlShiftD</td>
<td>CtrlShiftQ</td>
</tr>
<tr>
<td>CtrlE</td>
<td>CtrlR</td>
<td>CtrlShiftE</td>
<td>CtrlShiftR</td>
</tr>
<tr>
<td>CtrlF</td>
<td>CtrlS</td>
<td>CtrlShiftF</td>
<td>CtrlShiftS</td>
</tr>
<tr>
<td>CtrlG</td>
<td>CtrlT</td>
<td>CtrlShiftG</td>
<td>CtrlShiftT</td>
</tr>
<tr>
<td>CtrlH</td>
<td>CtrlU</td>
<td>CtrlShiftH</td>
<td>CtrlShiftU</td>
</tr>
<tr>
<td>CtrlI</td>
<td>CtrlV</td>
<td>CtrlShiftI</td>
<td>CtrlShiftV</td>
</tr>
<tr>
<td>CtrlJ</td>
<td>CtrlW</td>
<td>CtrlShiftJ</td>
<td>CtrlShiftW</td>
</tr>
<tr>
<td>CtrlK</td>
<td>CtrlX</td>
<td>CtrlShiftK</td>
<td>CtrlShiftX</td>
</tr>
<tr>
<td>CtrlL</td>
<td>CtrlY</td>
<td>CtrlShiftL</td>
<td>CtrlShiftY</td>
</tr>
<tr>
<td>CtrlM</td>
<td>CtrlZ</td>
<td>CtrlShiftM</td>
<td>CtrlShiftZ</td>
</tr>
</tbody>
</table>

The values of these enumeration members are equal to the `Keys` enumeration values (covered in Chapter 6) as they are combined in the `KeyData` property of the `KeyEventArgs` class. The following table shows shortcuts that consist of the number keys in combination with the Ctrl or Alt key:

### Shortcut Enumeration (selection)

<table>
<thead>
<tr>
<th>Ctrl0</th>
<th>CtrlShift0</th>
<th>Alt0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl1</td>
<td>CtrlShift1</td>
<td>Alt1</td>
</tr>
<tr>
<td>Ctrl2</td>
<td>CtrlShift2</td>
<td>Alt2</td>
</tr>
<tr>
<td>Ctrl3</td>
<td>CtrlShift3</td>
<td>Alt3</td>
</tr>
<tr>
<td>Ctrl4</td>
<td>CtrlShift4</td>
<td>Alt4</td>
</tr>
</tbody>
</table>
Shortcut Enumeration (selection)

<table>
<thead>
<tr>
<th>Ctrl0</th>
<th>CtrlShift0</th>
<th>Alt0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl5</td>
<td>CtrlShift5</td>
<td>Alt5</td>
</tr>
<tr>
<td>Ctrl6</td>
<td>CtrlShift6</td>
<td>Alt6</td>
</tr>
<tr>
<td>Ctrl7</td>
<td>CtrlShift7</td>
<td>Alt7</td>
</tr>
<tr>
<td>Ctrl8</td>
<td>CtrlShift8</td>
<td>Alt8</td>
</tr>
<tr>
<td>Ctrl9</td>
<td>CtrlShift9</td>
<td>Alt9</td>
</tr>
</tbody>
</table>

It's not possible to use a menu shortcut value that isn't defined in the Shortcut enumeration.

In Win32 programming, accelerator keys aren't restricted to menu items. In Windows Forms programming, however, shortcuts are always associated with menu items. If you want to define a shortcut that isn't associated with a menu item, you can define a menu item with that shortcut, include that menu item in your menu, and simply set the Visible property of the menu item to false. The item won't be displayed, but the shortcut will still be active.

**Your First Menu**

We now know enough to create our first menu. Unfortunately, we don't know quite enough to make the menu items very useful, such as displaying dialog boxes or using the clipboard, but it's a start. The Click event handlers in this program mostly just display message boxes to indicate that the menu item has been clicked.

**FirstMainMenu.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class FirstMainMenu : Form
{
    public static void Main()
    {
        Application.Run(new FirstMainMenu());
    }
    public FirstMainMenu()
    {
        Text = "First Main Menu";

        // Items on File submenu
        MenuItem miOpen = new MenuItem("&Open...",
            new EventHandler(MenuFileOpenOnClick),
            Shortcut.CtrlO);
        miOpen.CheckOnClick = true;
        MenuStrip m = new MenuStrip();
        m.Items.Add(miOpen);

        MenuItem miSave = new MenuItem("&Save",
            new EventHandler(MenuFileSaveOnClick),
            Shortcut.CtrlS);
        miSave.CheckOnClick = true;
        MenuStrip m = new MenuStrip();
        m.Items.Add(miSave);
    }
}
```
Shortcut.CtrlS);

MenuItem miSaveAs = new MenuItem("Save &As...",
    new EventHandler(MenuFileSaveAsOnClick));

MenuItem miDash = new MenuItem("-");

MenuItem miExit = new MenuItem("E&xit",
    new EventHandler(MenuFileExitOnClick));

// File item

MenuItem miFile = new MenuItem("&File",
    new MenuItem[] {miOpen, miSave,
                     miSaveAs,
                     miDash, miExit });

// Items on Edit submenu

MenuItem miCut = new MenuItem("Cu&t",
    new EventHandler(MenuEditCutOnClick),
    Shortcut.CtrlX);

MenuItem miCopy = new MenuItem("&Copy",
    new EventHandler(MenuEditCopyOnClick),
    Shortcut.CtrlC);

MenuItem miPaste = new MenuItem("&Paste",
    new EventHandler(MenuEditPasteOnClick),
    Shortcut.CtrlV);

// Edit item

MenuItem miEdit = new MenuItem("&Edit",
    new MenuItem[] {miCut, miCopy,
                   miPaste});

// Item on Help submenu

MenuItem miAbout = new MenuItem("&About FirstMainMenuItem...",
    new EventHandler(MenuHelpAboutOnClick));

// Help item

MenuItem miHelp = new MenuItem("&Help",
    new MenuItem[] {miAbout});

// Main menu
Menu = new MainMenu(new MenuItem[]{miFile, miEdit, miHelp});
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("File Open item clicked!", Text);
}
void MenuFileSaveOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("File Save item clicked!", Text);
}
void MenuFileSaveAsOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("File Save As item clicked!", Text);
}
void MenuFileExitOnClick(object obj, EventArgs ea)
{
    Close();
}
void MenuEditCutOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Edit Cut item clicked!", Text);
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Edit Copy item clicked!", Text);
}
void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Edit Paste item clicked!", Text);
}
void MenuHelpAboutOnClick(object obj, EventArgs ea)
{
    MessageBox.Show(Text + " © 2001 by Charles Petzold");
}

That this is a lot of code for a program that displays only a few message boxes I won't deny. But considering the overall importance of a menu to most applications, the amount of code isn't unreasonable. Much of what your program does it will do in response to menu Click events.

The program demonstrates the use of four different MenuItem constructors. For the Open menu item on the File menu, the program uses the constructor with three arguments:
MenuItem miOpen = new MenuItem("&Open...",
    new EventHandler(MenuFileOpenOnClick),
    Shortcut.CtrlO);

The Save As item on the File menu usually doesn't have a shortcut, so that constructor has only two arguments:
MenuItem miSaveAs = new MenuItem("Save &As...",
    new EventHandler(MenuFileSaveAsOnClick));

And here's a constructor that has only a text argument:
MenuItem miDash = new MenuItem("-");

When you specify a dash as the menu item in a submenu, a horizontal line is drawn in the submenu. This is the way you separate groups of items on a submenu.

When all the menu items on the File submenu have been created, the program creates a top-level menu item using yet another form of the MenuItem constructor:
MenuItem miFile = new MenuItem("&File",
    new MenuItem[] {miOpen, miSave, miSaveAs,
        miDash, miExit });

This constructor indicates the text of the item ("File") and an array of items that appear in the submenu invoked by that item.

The program continues with the Edit item (which invokes a submenu containing Cut, Copy, and Paste) and a Help item (containing just an About item). Finally, the form's constructor sets the Menu property of the form to an object of type MainMenu. The MainMenu constructor specifies an array of MenuItem objects that appear on the top level of the menu:
Menu = new MainMenu(new MenuItem[] {miFile, miEdit, miHelp});

All the Click event handlers in FirstMainMenu display message boxes except the event handler for the Exit item on the File menu. That event handler calls the Close method of Form to close the form and terminate the program.

You don't need to have separate event handlers for each Click event. You could handle every menu item in the same event handler. But using separate event handlers is cleaner and probably easier to maintain. The only time it makes sense to handle multiple items in the same Click event handler is when they form part of a related group, usually referring to mutually exclusive options.

Unconventional Menus

Although designing your menu to look like the menus of other Windows programs is helpful to your user, it's not mandatory. Sometimes—perhaps in a quickie program you're writing for yourself—you need only one menu item. If a top-level menu item doesn't invoke a submenu but instead carries out some action itself, it's customary to use an exclamation point to indicate that fact.

Here's a program that subclasses the JeuDeTaquin program from Chapter 10, "The Timer and Time," to add a one-item menu with the command "Scramble!"

JeuDeTaquinWithScramble.cs

// JeaDeTaquinWithScramble.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class JeuDeTaquinWithScramble: JeuDeTaquin
{
    public new static void Main()
    {
        Application.Run(new JeuDeTaquinWithScramble());
    }
}
public JeuDeTaquinWithScramble()
{
    Menu = new MainMenu(new MenuItem[] {
        new MenuItem("&Scramble!",
            new EventHandler(MenuScrambleOnClick)) });
}
void MenuScrambleOnClick(object obj, EventArgs ea)
{
    Randomize();
}
}

Now you don't have to end the program to rescramble the tiles.

One of the first popular programs to use a visual hierarchical menu in the character-mode MS-DOS environment was Lotus 1-2-3. The 1-2-3 menus didn't have pop-up menus, however. The menu display was restricted to a single line, and each level of menu replaced the one above it.

You can simulate an arrangement like that in Windows Forms by defining multiple MainMenu objects and then dynamically setting them to the Menu property of your form. Here's a program that demonstrates this technique.

OldFashionedMenu.cs
polator 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class OldFashionedMenu: Form
{
    MainMenu mmMain, mmFile, mmEdit;

    public static void Main()
    {
        Application.Run(new OldFashionedMenu());
    }
    public OldFashionedMenu()
    {
        Text = "Old-Fashioned Menu";
        EventHandler eh = new EventHandler(MenuOnClick);

        mmMain = new MainMenu(new MenuItem[]
            {
                new MenuItem("MAIN:"),
                new MenuItem("&File", new EventHandler(MenuFileOnClick)),
                new MenuItem("&Edit", new EventHandler(MenuEditOnClick))
            } );
}
void MenuFileOnClick(object obj, EventArgs ea)
{
    // File menu commands here
}
void MenuEditOnClick(object obj, EventArgs ea)
{
    // Edit menu commands here
}
void MenuOnClick(object obj, EventArgs ea)
{
    // Main menu commands here
}
}
mmFile = new MainMenu(new MenuItem[]
{
    new MenuItem("FILE:"),
    new MenuItem("&New", eh),
    new MenuItem("&Open...", eh),
    new MenuItem("&Save", eh),
    new MenuItem("Save &As...", eh),
    new MenuItem("(&Main)", new EventHandler(MenuMainOnClick))
});

mmEdit = new MainMenu(new MenuItem[]
{
    new MenuItem("EDIT:"),
    new MenuItem("Cu&t", eh),
    new MenuItem("&Copy", eh),
    new MenuItem("&Paste", eh),
    new MenuItem("De&lete", eh),
    new MenuItem("(&Main)", new EventHandler(MenuMainOnClick))
});

Menu = mmMain;
}

void MenuMainOnClick(object obj, EventArgs ea)
{
    Menu = mmMain;
}

void MenuFileOnClick(object obj, EventArgs ea)
{
    Menu = mmFile;
}

void MenuEditOnClick(object obj, EventArgs ea)
{
    Menu = mmEdit;
}

void MenuOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Menu item clicked!", Text);
}

**MenuItem Properties and Events**

The one `MenuItem` constructor (of the five I originally listed) that I haven't demonstrated is the default constructor:

```
new MenuItem()
```
If you use this constructor, you must then have statements that set properties of the `MenuItem` object. The properties connected with the menu item text and shortcut are listed in this table:

### MenuItem Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
<tr>
<td>char</td>
<td>Mnemonic</td>
<td>get</td>
</tr>
<tr>
<td>Shortcut</td>
<td>Shortcut</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>ShowShortcut</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `Mnemonic` property is the character that follows the ampersand in the `Text` property (or 0 if there is no mnemonic character). You can set the `ShowShortcut` property to `false` to inhibit the display of the shortcut to the right of the menu item text.

The following list shows most of the read/write `bool` properties of `MenuItem`:

### MenuItem Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Visible</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Enabled</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>DefaultItem</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Break</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>BarBreak</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Setting the `Visible` property to `false` causes the menu item to not appear in the menu. However, the shortcut (if any) still invokes the `Click` event handler. You can use this property to set shortcuts that aren't associated with any menu items. You can also make a number of menu items optionally invisible to implement a simplified-menu system that won't overwhelm beginners.

The `Enabled` property is probably the most commonly used of these five properties. It is often set to `false` to disable menu items that aren't currently applicable. When an item is disabled, the text is displayed in a weak font that indicates the item is unavailable. You can't trigger the `Click` event handler for a disabled item, either by clicking the item or by typing the keyboard shortcut.

Items on the File and Edit menus are often enabled and disabled based on certain conditions. The Save and Save As options are typically disabled if the program currently has no document loaded. A program disables the Cut and Copy options if a document has been loaded but nothing is currently selected in the document. A program disables Paste if the clipboard currently has nothing the program can use. I'll explain how to handle the disabling of menu items later in this chapter.

When you set the `DefaultItem` property to `true`, the menu item is displayed in boldface. For a program's main menu, the `DefaultItem` property makes sense only for items on submenus. When you double-click the item that invokes the submenu, the default item is triggered. For example, if you insert the statement

```csharp
miExit.DefaultItem = true;
```

in the constructor of `FirstMainMenu`, double-clicking the File item causes the program to terminate. Default items are more common on context menus.

Setting the `Break` property to `true` causes the menu item (and subsequent menu items) to be displayed in a new column. Setting the `BarBreak` property to `true` has the same effect as `Break` but also draws a vertical line between the columns. Although it's not common, you can use `Break` and `BarBreak` with items on the top level of a main menu. Both properties have the same effect of displaying the menu item (and subsequent menu items) in a new row.
These are not all the properties of MenuItem. I'll get to the Checked and RadioCheck properties shortly. In addition, the MenuItem class—as well as MainMenu and ContextMenu—inherits from Menu an extremely important class named MenuItems (notice the plural) that I'll also talk about later in this chapter.

The MenuItem class defines five events, of which Click is obviously the most crucial. Two of the five events refer to the owner-draw facility of menus, which I'll discuss toward the end of this chapter. The other three events (including Click) are shown here:

**MenuItem Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>OnClick</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Popup</td>
<td>OnPopup</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>Select</td>
<td>OnSelect</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

Programs often install Popup event handlers for top-level items such as File or Edit. The Popup event tells you when the submenu associated with that top-level item is about to be displayed. As I mentioned earlier, some menu items, such as Cut, Copy, and Paste, must be enabled or disabled based on whether something has been selected in the document or whether the clipboard contains data the application can use. The Popup event handler is the perfect opportunity to perform this enabling and disabling of items.

The Select event occurs when the mouse cursor passes over a menu item or the user presses the arrow keys to move among menu items. The selected menu item is usually displayed in a different color. As you may know, some applications use a status bar to display a simple text description of each menu item as it's being selected. I'll demonstrate how to do this in Chapter 20.

**Checking the Items**

If you look at the View menu of the Windows Calculator (and particularly if you switch to the Scientific format), you'll see several examples of menu items that are checked. Menu items that represent Boolean choices—such as the Digit Grouping item—can be checked or unchecked just like a CheckBox control.

Other groups of menu items, such as the Hex, Decimal, Octal, and Binary items in Calculator, represent mutually exclusive options. The currently selected item is indicated by a filled circle, called a radio check. (Some Windows programs use the regular menu check mark for mutually exclusive menu items as well as on-off items. The check mark is allowed but no longer encouraged for mutually exclusive items.)

You control the check mark and radio check with the following two properties of MenuItem:

**MenuItem Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Checked</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>RadioCheck</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Set the Checked property to true if the mark (regardless whether it's a check or a circle) is to be displayed. Set RadioCheck to true to display a circle (indicating mutually exclusive options) or false for check marks (for on-off items).

Here's a program similar to the RadioButtons program of Chapter 12. The main menu contains a single item named Format that contains ten items—eight items are used like radio buttons to select a color; the ninth item is a horizontal bar; and the tenth is an item with the text Fill that can be checked to indicate that the ellipse should be filled.

**CheckAndRadioCheck.cs**

```csharp
//-------------------------------------------------
// CheckAndRadioCheck.cs © 2001 by Charles Petzold
//-------------------------------------------------
```
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckAndRadioCheck : Form
{
    MenuItem miColor, miFill;

    public static void Main()
    {
        Application.Run(new CheckAndRadioCheck());
    }
    public CheckAndRadioCheck()
    {
        Text = "Check and Radio Check";
        ResizeRedraw = true;

        string[] astrColor = {"Black", "Blue", "Green", "Cyan", "Red", "Magenta", "Yellow", "White"};
        MenuItem[] ami = new MenuItem[astrColor.Length + 2];
        EventHandler ehColor = new EventHandler(MenuFormatColorOnClick);
        for (int i = 0; i < astrColor.Length; i++)
        {
            ami[i] = new MenuItem(astrColor[i], ehColor);
            ami[i].RadioCheck = true;
        }
        miColor = ami[0];
        miColor.Checked = true;
        ami[astrColor.Length] = new MenuItem("-");
        miFill = new MenuItem("&Fill",
            new EventHandler(MenuFormatFillOnClick));
        ami[astrColor.Length + 1] = miFill;
        MenuItem mi = new MenuItem("&Format", ami);
        Menu = new MainMenu(new MenuItem[] {mi});
    }
    void MenuFormatColorOnClick(object obj, EventArgs ea)
    {
        miColor.Checked = false;
    }
    void MenuFormatFillOnClick(object obj, EventArgs ea)
    {
    }
}
miColor = (MenuItem)obj;
miColor.Checked = true;

Invalidate();
}
void MenuFormatFillOnClick(object obj, EventArgs ea)
{
    MenuItem mi = (MenuItem)obj;
    mi.Checked ^= true;
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    if (miFill.Checked)
    {
        Brush brush = new SolidBrush(Color.FromName(miColor.Text));
        grfx.FillEllipse(brush, 0, 0, ClientSize.Width - 1,
                         ClientSize.Height - 1);
    }
    else
    {
        Pen pen = new Pen(Color.FromName(miColor.Text));
        grfx.DrawEllipse(pen, 0, 0, ClientSize.Width - 1,
                         ClientSize.Height - 1);
    }
}

The constructor defines an array of eight text strings representing colors and then an array of MenuItem structures sufficient to accommodate those eight colors and two more menu items:

MenuItem[] ami = new MenuItem[astrColor.Length + 2];

A group of mutually exclusive menu items are generally associated with the same Click event handler. For that reason, the event handler is defined before any of the menu items are created:

EventHandler ehColor = new EventHandler(MenuFormatColorOnClick);

A for loop creates the eight menu items based on the eight color names and the ehColor event handler. The RadioCheck property is set to true so that a circle is displayed rather than a check mark when the Checked property is set to true.

The miColor variable stored as a field is the MenuItem object that is currently checked. The constructor sets this field to the first MenuItem in the array and then sets the Checked property to true.

miColor = ami[0];
miColor.Checked = true;
Let me go over the `MenuFormatColorOnClick` handler now and then come back to the program's constructor. The `Click` event handler begins by unchecking the currently checked menu item:

```csharp
miColor.Checked = false;
```

The `miColor` field is then set to the first argument of the event handler, which is the item the user has clicked:

```csharp
miColor = (MenuItem)obj;
```

The event handler then checks the menu item

```csharp
miColor.Checked = true;
```

and invalidates the form to repaint the ellipse. This block of code demonstrates the customary way to check and uncheck mutually exclusive menu items.

Let's return to the constructor. After creating the eight menu items for the eight colors, it creates a ninth menu item that is a horizontal dividing line and then a tenth menu item for the Fill item:

```csharp
miFill = new MenuItem("&Fill",
                        new EventHandler(MenuFormatFillOnClick));
ami[ astrColor.Length + 1 ] = miFill;
```

The `miFill` variable is also stored as a field. The `OnPaint` method uses both `miColor` and `miFill` to draw (or fill) the ellipse.

The `MenuFormatFillOnClick` method doesn't need to access `miFill`, however. The event handler obtains the `MenuItem` object being clicked by casting the first argument,

```csharp
MenuItem mi = (MenuItem)obj;
```

and then toggles the state of that item:

```csharp
mi.Checked ^= true;
```

You could replace these two statements with the single statement

```csharp
miFill.Checked ^= true;
```

but the event handler is more generalized if it doesn't refer to a specific menu item. If you added other on-off menu items that affected `OnPaint` processing, you could use the same general-purpose checking-and-unchecking method.

The `OnPaint` method in `CheckAndRadioCheck` converts the text color name from the menu item into a `Color` object by using the static method `Color.FromName` in preparation for creating a brush or pen:

```csharp
Color.FromName(miColor.Text)
```

Obviously, not every menu item has a `Text` property that can be directly converted into a usable object. Using menu text in this way isn't a good idea for a couple reasons. First, if the menu needs to be translated into another language, the new text might not convert to `Color` objects so readily. Second, it's awkward to put ampersands in the color names because they would have to be stripped out before passing the text to the `Color.FromName` method.

I'll demonstrate a more generalized approach to differentiating mutually exclusive menu items shortly. (Does it involve deriving a class from `MenuItem`? you ask. What do you think?)

## Working with Context Menus

Context menus are in some ways simpler than main menus, mostly because they are smaller, sometimes containing only a list of menu items without any submenus. The following program creates a context menu that lets you select the background color of the form.

```csharp
ContextMenuDemo.cs
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ContextMenuDemo : Form
{
    MenuItem miColor;

    public static void Main()
    {
        Application.Run(new ContextMenuDemo());
    }
    public ContextMenuDemo()
    {
        Text = "Context Menu Demo";
        EventHandler eh = new EventHandler(MenuColorOnClick);
        MenuItem[] ami = { new MenuItem("Black", eh),
                          new MenuItem("Blue", eh),
                          new MenuItem("Green", eh),
                          new MenuItem("Cyan", eh),
                          new MenuItem("Red", eh),
                          new MenuItem("Magenta", eh),
                          new MenuItem("Yellow", eh),
                          new MenuItem("White", eh) };;

        foreach (MenuItem mi in ami)
            mi.RadioCheck = true;

        miColor = ami[3];
        miColor.Checked = true;
        BackColor = Color.FromName(miColor.Text);

        ContextMenu = new ContextMenu(ami);
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        miColor.Checked = false;
        miColor = (MenuItem) obj;
        miColor.Checked = true;
        BackColor = Color.FromName(miColor.Text);
    }
}
In this program, the eight menu items are defined right in the initialization of the MenuItem array named ami. As in the previous program, all the menu items use the same Click event handler. A for loop sets the RadioCheck property of each menu item to true. The constructor then sets the field variable miColor to the fourth menu item in the array, checks that item, and sets the background color to the checked item.

The constructor concludes by making a new ContextMenu object from the array of MenuItem objects and then assigning that to the ContextMenu property of the form:

```csharp
ContextMenu = new ContextMenu(ami);
```

You can invoke the context menu by right-clicking anywhere within the client area. Alternatively, the ContextMenu class also has a method that lets you display a context menu without setting the ContextMenu property of a control:

```
ContextMenu Methods (selection)

void Show(Control ctrl, Point ptLocation)
```

You can use this method if you need to display different context menus for a single control (or form) depending on where the mouse is clicked.

By converting the menu item text to a color, the ContextMenuDemo program has the same flaws as CheckAndRadioCheck. When implementing mutually exclusive menu items that use the same Click event handler, a much better (and more generalized) approach is to derive a class from MenuItem and use that class in your menu instead. The class derived from MenuItem includes a new field or property that stores an object to identify the item.

Here's a program that derives a class named MenuItemColor from MenuItem. The class includes a private field named clr to store a Color object. The public property named Color gives other classes access to that color. In addition, the new class also includes a new constructor that lets a MenuItemColor object be created with a specified color. The following program is very similar to ContextMenuDemo except that it uses MenuItemColor rather than MenuItem.

```csharp
BetterContextMenu.cs
//-------------------------------------------------
// BetterContextMenu.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterContextMenu: Form
{
    MenuItemColor micColor;

    public static void Main()
    {
        Application.Run(new BetterContextMenu());
    }

    public BetterContextMenu()
    {
```
Text = "Better Context Menu Demo";

EventHandler eh = new EventHandler(MenuColorOnClick);

MenuItemColor[] amic =
{
    new MenuItemColor(Color.Black,   "&Black",   eh),
    new MenuItemColor(Color.Blue,    "B&lue",    eh),
    new MenuItemColor(Color.Green,   "&Green",   eh),
    new MenuItemColor(Color.Cyan,    "&Cyan",    eh),
    new MenuItemColor(Color.Red,     "&Red",     eh),
    new MenuItemColor(Color.Magenta, "&Magenta", eh),
    new MenuItemColor(Color.Yellow,  "&Yellow",  eh),
    new MenuItemColor(Color.White,   "&White",   eh)
};

foreach (MenuItemColor mic in amic)
    mic.RadioCheck = true;

micColor = amic[3];
micColor.Checked = true;
BackColor = micColor.Color;

ContextMenu = new ContextMenu(amic);
}

void MenuColorOnClick(object obj, EventArgs ea)
{
    micColor.Checked = false;
    micColor = (MenuItemColor) obj;
    micColor.Checked = true;

    BackColor = micColor.Color;
}
}

class MenuItemColor: MenuItem
{
    Color clr;

    public MenuItemColor(Color clr, string str, EventHandler eh):
        base(str, eh)
    {
        Color = clr;
    }

    public Color Color
    {
    }
Now the program can set the `BackColor` property of the form directly from the `Color` property of the currently checked `MenuItemColor` object.

### The Menu Item Collection

If you look back at the FirstMainMenu program, you'll see that the menu was built from the inside out. It began by creating `MenuItem` objects for the innermost items (such as Open, Save, and so on). These were assembled into arrays to create top-level items (File, Edit, and so forth). Then the top-level menu items were gathered together into a `MainMenu` object.

A program might be more coherent and maintainable if the menu were created from the top down, beginning by creating a `MainMenu` object, adding `MenuItem` objects to the main menu to create top-level items (File, Edit, and so forth), and then adding other items in the submenus (Open, Save, and so forth).

This second approach is made possible by a couple important properties defined in the `Menu` class. As I mentioned early in this chapter, the `MenuItem`, `MainMenu`, and `ContextMenu` classes are all derived from `Menu`, so they all inherit these properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td><code>IsParent</code></td>
<td><code>get</code></td>
</tr>
<tr>
<td><code>Menu.MenuItemCollection</code></td>
<td><code>MenuItems</code></td>
<td><code>get</code></td>
</tr>
</tbody>
</table>

Does that `Menu.MenuItemCollection` type look like a familiar friend yet? It's quite similar to the `ImageList.ImageCollection` class in Chapter 11 and the `Control.ControlCollection` class in Chapter 12. Like those other classes, `Menu.MenuItemCollection` implements the `IList`, `ICollection`, and `IEnumerable` interfaces. You can index the `MenuItems` property as if it were an array of `MenuItem` objects. You can also call methods named `Add`, `Remove`, and `Clear`.

The `IsParent` property indicates that a menu item is parent to other menu items and hence the `MenuItems` property is valid.

In addition, the `MenuItem` class has the following two related properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Menu</code></td>
<td><code>Parent</code></td>
<td><code>get</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Index</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

The `Parent` property indicates the parent menu of a particular menu item; the `Index` property (which is also writable) indicates the zero-based index of a `MenuItem` object within a particular submenu.

The `Menu.MenuItemCollection` class implements the following methods that let you add child menu items to a main menu, a context menu, or another menu item:

#### `Menu.MenuItemCollection` Methods (selection)

- `MenuItem Add(string strText)`
- `MenuItem Add(string strText, EventHandler ehClick)`
- `MenuItem Add(string strText, MenuItem[] ami)`
- `int Add(MenuItem mi)`
- `int Add(int index, MenuItem mi)`
void AddRange(MenuItem[] ami)

A program can use the following properties (the second of which is an indexer) to determine how many menu items the collection contains and to obtain them:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Count</td>
<td>get</td>
</tr>
<tr>
<td>MenuItem</td>
<td>()</td>
<td>get</td>
</tr>
</tbody>
</table>

Menu items can also be located and removed:

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool Contains(MenuItem mi)</td>
</tr>
<tr>
<td>int IndexOf(MenuItem mi)</td>
</tr>
<tr>
<td>void Remove(MenuItem mi)</td>
</tr>
<tr>
<td>void RemoveAt(int index)</td>
</tr>
<tr>
<td>void Clear()</td>
</tr>
</tbody>
</table>

Here’s a version of the ContextMenuDemo program, named ContextMenuAdd, that uses the Add and indexing facility of the MenuItem property to create the menu. The ContextMenu object is created first, and then the menu items are added to it.

ContextMenuAdd.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ContextMenuAdd : Form
{
    MenuItem miColor;

    public static void Main()
    {
        Application.Run(new ContextMenuAdd());
    }

    public ContextMenuAdd()
    {
        Text = "Context Menu Using Add";

        ContextMenu cm = new ContextMenu();
        EventHandler eh = new EventHandler(MenuColorOnClick);
```
cm.MenuItems.Add("Black", eh);
cm.MenuItems.Add("Blue", eh);
cm.MenuItems.Add("Green", eh);
cm.MenuItems.Add("Cyan", eh);
cm.MenuItems.Add("Red", eh);
cm.MenuItems.Add("Magenta", eh);
cm.MenuItems.Add("Yellow", eh);
cm.MenuItems.Add("White", eh);

foreach (MenuItem mi in cm.MenuItems)
    mi.RadioCheck = true;

miColor = cm.MenuItems[3];
miColor.CheckedChanged = true;
BackColor = Color.FromName(miColor.Text);

ContextMenu = cm;
}
void MenuColorOnClick(object obj, EventArgs ea)
{
    miColor.CheckedChanged = false;
    miColor = (MenuItem) obj;
    miColor.CheckedChanged = true;

    BackColor = Color.FromName(miColor.Text);
}

This program could have saved a statement by assigning the new ContextMenu object to the ContextMenu property of the form:
ContextMenu = new ContextMenu();

The statements adding the items would then look like this:
ContextMenu.MenuItems.Add("Black", eh);

Earlier in this chapter, I transformed the ContextMenuDemo program into the BetterContextMenu program by defining a class named MenuItemColor that inherits from MenuItem but also stores a Color object. What would I need to do to convert the ContextMenuAdd program to use MenuItemColor objects?

What's most obvious is that I couldn't use the same Add method I used in ContextMenuAdd. The method call
cm.MenuItems.Add("Black", eh);
implicitly creates an object of type MenuItem and then adds it to the menu item collection. The following statement does the same job more explicitly:

To convert the program to use the MenuItemColor class, you'd need to make calls like so:

```csharp
cm.MenuItems.Add(new MenuItemColor(Color.Black, "Black", eh));
```
The next program has a single top-level menu item named Facename and uses the *Popup* event as an opportunity to add all the available font facenames to the menu. In theory, this approach is better than building the menu when the program starts up because the program can display fonts that are added after the program starts running.

**FontMenu.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class FontMenu: Form {
    const int iPointSize = 24;
    string    strFacename;

    public static void Main() {
        Application.Run(new FontMenu());
    }
    public FontMenu() {
        Text = "Font Menu";
        strFacename = Font.Name;
        Menu = new MainMenu();
        MenuItem mi = new MenuItem("&Facename");
        mi.Popup += new EventHandler(MenuFacenameOnPopup);
        mi.MenuItems.Add(" ");  // Necessary for pop-up call
        Menu.MenuItems.Add(mi);
    }
    void MenuFacenameOnPopup(object obj, EventArgs ea) {
        MenuItem     miFacename = (MenuItem)obj;
        FontFamily[] aff        = FontFamily.Families;
        EventHandler ehClick    = new EventHandler(MenuFacenameOnClick);
        MenuItem[] ami        = new MenuItem[aff.Length];
        for (int i = 0; i < aff.Length; i++)
            
    // Necessary for pop-up call
}
if (aff[i].Name == strFacename)
    ami[i].Checked = true;
}
miFacename.MenuItems.Clear();
miFacename.MenuItems.AddRange(ami);
}

void MenuFacenameOnClick(object obj, EventArgs ea)
{
    MenuItem mi = (MenuItem)obj;
    strFacename = mi.Text;
    Invalidate();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Font    font = new Font(strFacename, iPointSize);
    StringFormat strfmt = new StringFormat();
    strfmt.Alignment     = StringAlignment.Center;
    strfmt.LineAlignment = StringAlignment.Center;
    grfx.DrawString("Sample Text", font, new SolidBrush(ForeColor),
        ClientRectangle, strfmt);
}

When the constructor defines the menu, it adds a single blank item to the submenu. At least one item in the submenu seems to be necessary to generate a Popup event.

The MenuFacenameOnPopup event handler begins by obtaining the top-level menu item:
MenuItem miFacename = (MenuItem)obj;

The method then obtains an array of all the available font facenames by calling the static FontFamily.Families method. It defines a MenuItem array of that size and then initializes all the entries, in the process setting the Check property of the item corresponding to the font facename currently stored in the strFacename field.

After the MenuItem array has been initialized, the method concludes by clearing all the existing entries from the pop-up menu and adding the new array of items:
miFacename.MenuItems.Clear();
miFacename.MenuItems.AddRange(ami);

To keep the code bulk down, this program doesn't have a fix for the problem that results when you pick a font facename that isn't capable of the FontStyle.Regular style (such as Aharoni). If you pick such a font, the Font constructor in the OnPaint method will throw an exception and terminate the program. (See Chapter 9 for more details on this issue and on working with fonts.)

But the real problem with this program is the size of the submenu. It's very likely that the list will exceed the height of your display. Picking a font facename is obviously a job for a dialog box, as I'll demonstrate in Chapter 16.

The Standard Menu (A Proposal)
To make your program easy to read and maintain, it seems logical that the flow of the menu-creation code should mimic the hierarchy of the menu. Thus, the constructor should probably build the menu starting with the first top-level item (typically File), then all the items in the File submenu (typically New, Open, Save, and so forth), then the next top-level item (typically Edit), then the items in the Edit submenu, and ending with the About item on the Help menu.

As you begin experimenting with top-down menu construction using the Add method of the Menu.MenuItemCollection class, you'll find that the Add methods are not created equal. Some are more useful than others. You may want to adopt a standard style for the creation of your main menu that makes use of just a small subset of the MenuItem constructors and the Add methods. The proposal I've outlined here uses just one form of MenuItem constructor and one form of the Add method. You're free to disregard this proposal. I disregard it myself in later chapters. But I'd like to explore some of the issues involved in menu construction code.

Let's first look at some top-down menu creation statements for nonfunctional items without any event handlers or shortcuts. The code would look something like this:

```csharp
Menu = new MainMenu();
Menu.MenuItems.Add("&File");
Menu.MenuItems[0].MenuItems.Add("&Open...");
Menu.MenuItems[0].MenuItems.Add("&Save...");
...
Menu.MenuItems.Add("&Edit");
Menu.MenuItems[1].MenuItems.Add("Cu&t");
Menu.MenuItems[1].MenuItems.Add("&Copy");
...
```

This code is all very orderly (if not exactly pretty): The top-level items (File and Edit) are added to the MainMenu object using the Add method of its MenuItems property. The expression `Menu.MenuItems[0]` refers to the File menu item, and `Menu.MenuItems[1]` refers to the Edit menu item. Each of those menu items has its own MenuItems property that is the collection of menu items on the submenu. You use the Add method of that MenuItems property to add items Open, Save, Cut, Copy, and so forth.

Except for groups of mutually exclusive menu items, most menu items should be associated with their own Click event handlers. To make the menu functional, you need to convert the previous statements into statements like this:

```csharp
Menu.MenuItems[0].MenuItems.Add("&Open...",
    new EventHandler(MenuFileOpenOnClick));
```

But the Open item also commonly includes a Ctrl+O shortcut, and there's no Add method that includes a shortcut argument. You'd need an additional statement like

```csharp
Menu.MenuItems[0].MenuItems[0].Shortcut = Shortcut.CtrlO;
```

to reference the Shortcut property of the Open menu item. But if you then modify your code to put a New item before the Open item, you need to change the statement so the indexing is different:

```csharp
```

We're headed down a wrong path with this approach. I think you'll agree that setting the property of a menu item shouldn't require going through two levels of MenuItems properties.

Probably a better approach is to define the MenuItem first, as here,

```csharp
miFileOpen = new MenuItem("&Open",
    new EventHandler(MenuFileOpenOnClick),
    Shortcut.CtrlO);
```

and then add this menu item to the MenuItems collection:

```csharp
Menu.MenuItems[0].MenuItems.Add(miFileOpen);
```
Since the creation of the \texttt{miFileOpen} object spills over beyond one line of code, it may be clearer if we go with a simple constructor and a more explicit assignment of the \texttt{MenuItem} properties:

\begin{verbatim}
miFileOpen = new MenuItem("&Open");
miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
miFileOpen.Shortcut = Shortcut.CtrlO;
Menu.MenuItems[0].MenuItems.Add(miFileOpen);
\end{verbatim}

Does it ever make sense to use the version of \texttt{Add} that has a single \texttt{string} argument? It's a good choice when you want to add a horizontal separation line to the menu:

\begin{verbatim}
Menu.MenuItems[0].MenuItems.Add("-");
\end{verbatim}

And this version of \texttt{Add} might make sense when adding a top-level item:

\begin{verbatim}
Menu.MenuItems.Add("F&ormat");
\end{verbatim}

But many top-level items should have \texttt{PopupMenu} event handlers installed to enable or disable menu items on the submenu. So here also, it makes more sense to create the \texttt{MenuItem} first, set its properties, and then add the menu item to the menu:

\begin{verbatim}
MenuItem mi = new MenuItem("&File");
mi.Popup += new EventHandler(MenuFileOnPopup);
Menu.MenuItems.Add(mi);
\end{verbatim}

If you want to be totally consistent throughout your menu creation, all you need to use is one form of the \texttt{MenuItem} constructor (the one with just a \texttt{string} argument) and one form of the \texttt{Add} method (the one with a \texttt{MenuItem} argument).

As you may have gathered from much of this book, I prefer to keep the number of field variables in a class to a minimum. The Menu Designer in Visual Studio .NET makes every \texttt{MenuItem} object a field. That's clearly unnecessary! Because most menu items have their own \texttt{Click} event handlers, it isn't necessary for the program to retain all the \texttt{MenuItem} objects. An exception is when items must be enabled or disabled during a \texttt{PopupMenu} event. Those items should probably be stored as fields.

Finally, I want to justify one more little tiny variable—just a little \texttt{int} that I call \texttt{index}. Every time you add an item to the top level of the menu with a statement like

\begin{verbatim}
Menu.MenuItems.Add(mi);
\end{verbatim}

you can also calculate a new \texttt{index} value:

\begin{verbatim}
index = Menu.MenuItems.Count - 1;
\end{verbatim}

Use this \texttt{index} value to add items to each of the submenus, as here:

\begin{verbatim}
Menu.MenuItems[index].MenuItems.Add(miFileOpen);
\end{verbatim}

For the File menu, the use of a variable rather than an explicit 0 is hardly necessary: File will always be the first item on the main menu, from now until the end of time. But the \texttt{index} variable makes loads of sense for later submenus, particularly if you someday revise your constructor code to insert a new submenu.

Here's a program that demonstrates my approach to creating a standard menu from the top down consistently using one form of the \texttt{MenuItem} constructor and one form of the \texttt{Add} method.

\texttt{StandardMenu.cs}

\begin{verbatim}
//------------------------------
// StandardMenu.cs © 2001 by Charles Petzold
//------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
\end{verbatim}
class StandardMenu: Form
{
    MenuItem miFileOpen, miFileSave;
    MenuItem miEditCut, miEditCopy, miEditPaste;

    // Experimental variables for Popup code
    bool bDocumentPresent = true;
    bool bNonNullSelection = true;
    bool bStuffInClipboard = false;

    public static void Main()
    {
        Application.Run(new StandardMenu());
    }
    public StandardMenu()
    {
        Text = "Standard Menu";
        Menu = new MainMenu();

        // File
        MenuItem mi = new MenuItem("&File");
        mi.Popup += new EventHandler(MenuFileOnPopup);
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

        // File Open
        miFileOpen = new MenuItem("&Open...");
        miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
        miFileOpen.Shortcut = Shortcut.CtrlO;
        Menu.MenuItems[index].MenuItems.Add(miFileOpen);

        // File Save
        miFileSave = new MenuItem("&Save");
        miFileSave.Click += new EventHandler(MenuFileSaveOnClick);
        miFileSave.Shortcut = Shortcut.CtrlS;
        Menu.MenuItems[index].MenuItems.Add(miFileSave);

        // Horizontal line
        mi = new MenuItem("-");
        Menu.MenuItems[index].MenuItems.Add(mi);
// File Exit

mi = new MenuItem("E&xit");
mi.Click += new EventHandler(MenuFileExitOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

// Edit

mi = new MenuItem("&Edit");
mi.Popup += new EventHandler(MenuEditOnPopup);
Menu.MenuItems.Add(mi);
index = Menu.MenuItems.Count - 1;

// Edit Cut

miEditCut = new MenuItem("Cu&t");
miEditCut.Click += new EventHandler(MenuEditCutOnClick);
miEditCut.Shortcut = Shortcut.CtrlX;
Menu.MenuItems[index].MenuItems.Add(miEditCut);

// Edit Copy

miEditCopy = new MenuItem("&Copy");
miEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
miEditCopy.Shortcut = Shortcut.CtrlC;
Menu.MenuItems[index].MenuItems.Add(miEditCopy);

// Edit Paste

miEditPaste = new MenuItem("&Paste");
miEditPaste.Click += new EventHandler(MenuEditPasteOnClick);
miEditPaste.Shortcut = Shortcut.CtrlV;
Menu.MenuItems[index].MenuItems.Add(miEditPaste);

// Help

mi = new MenuItem("&Help");
Menu.MenuItems.Add(mi);
index = Menu.MenuItems.Count - 1;

// Help About

mi = new MenuItem("&About StandardMenu...");
mi.Click += new EventHandler(MenuHelpAboutOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

void MenuFileOnPopup(object obj, EventArgs ea)
{
    miFileSave.Enabled = bDocumentPresent;
}

void MenuEditOnPopup(object obj, EventArgs ea)
{
    miEditCut.Enabled = bNonNullSelection;
    miEditCopy.Enabled = bNonNullSelection;
    miEditPaste.Enabled = bStuffInClipboard;
}

void MenuFileOpenOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("This should be a File Open dialog box!", Text);
}

void MenuFileSaveOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("This should be a File Save dialog box!", Text);
}

void MenuFileExitOnClick(object obj, EventArgs ea)
{
    Close();
}

void MenuEditCutOnClick(object obj, EventArgs ea)
{
    // Copy selection to clipboard; delete from document.
}

void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    // Copy selection to clipboard.
}

void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    // Copy clipboard data to document.
}

void MenuHelpAboutOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("StandardMenu © 2001 by Charles Petzold", Text);
}

Although this is certainly not the tersest code imaginable, I think the program achieves a significant
degree of clarity and maintainability.

The Owner-Draw Option
We're nearing the end of this chapter and I still haven't shown you how to put little pictures in your menu items or how to use a different font or different colors.

Any menu feature beyond what I've covered so far requires that you use a facility called owner-draw. For every MenuItem object that you want to draw yourself, you must set the following property to true:

**MenuItem Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>OwnerDraw</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Usually you'd set this property only for items on pop-up menus. If you set OwnerDraw to true, you must also install event handlers for the following two events:

**MenuItem Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>MeasureItem</td>
<td>OnMeasureItem</td>
<td>MeasureItemEventHandler</td>
<td>MeasureItemEventArgs</td>
</tr>
<tr>
<td>DrawItem</td>
<td>OnDrawItem</td>
<td>DrawItemEventHandler</td>
<td>DrawItemEventArgs</td>
</tr>
</tbody>
</table>

Whenever Windows is preparing to draw a menu item (which is usually when it's preparing to display a pop-up menu), it calls the handler for the MeasureItem event. The event is accompanied by an object of type MeasureItemEventArgs.

**MeasureItemEventArgs Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Index</td>
<td>get</td>
</tr>
<tr>
<td>Graphics</td>
<td>Graphics</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>ItemWidth</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>ItemHeight</td>
<td>get/set</td>
</tr>
</tbody>
</table>

On entry to the MeasureItem event handler, the ItemWidth and ItemHeight properties are set to 0. Your responsibility is to set them to the total width and height of the menu item you intend to draw. The Index property is there to help your event handler figure out which item requires measurement. If necessary, the Graphics property lets you obtain the device resolution in dots per inch, or the size of text items by calling MeasureString.

A short time later, Windows calls the DrawItem event handler, accompanied by an object of DrawItemEventArgs:

**DrawItemEventArgs Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Index</td>
<td>get</td>
</tr>
<tr>
<td>Graphics</td>
<td>Graphics</td>
<td>get</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Bounds</td>
<td>get</td>
</tr>
<tr>
<td>DrawItemState</td>
<td>State</td>
<td>get</td>
</tr>
<tr>
<td>Font</td>
<td>Font</td>
<td>get</td>
</tr>
<tr>
<td>Color</td>
<td>BackColor</td>
<td>get</td>
</tr>
<tr>
<td>Color</td>
<td>ForeColor</td>
<td>get</td>
</tr>
</tbody>
</table>

Your program's responsibility is to draw the item using the Graphics object within the rectangle defined by the Bounds property. Don't assume that the Bounds property has an upper left corner at point (0, 0)! In fact, the Bounds rectangle is a rectangle within the entire pop-up menu.
The width of the \textit{Bounds} rectangle will be greater than the amount you specified while handling the \textit{MeasureItem} event to allow for a check mark of standard size at the left of the item.

The \textit{DrawItemState} enumeration tells you whether the item is selected, disabled, or checked:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>Selected</td>
<td>1</td>
</tr>
<tr>
<td>Grayed</td>
<td>2</td>
</tr>
<tr>
<td>Disabled</td>
<td>4</td>
</tr>
<tr>
<td>Checked</td>
<td>8</td>
</tr>
<tr>
<td>Focus</td>
<td>16</td>
</tr>
<tr>
<td>Default</td>
<td>32</td>
</tr>
<tr>
<td>HotLight</td>
<td>64</td>
</tr>
<tr>
<td>Inactive</td>
<td>128</td>
</tr>
<tr>
<td>NoAccelerator</td>
<td>256</td>
</tr>
<tr>
<td>NoFocusRect</td>
<td>512</td>
</tr>
<tr>
<td>ComboBoxEdit</td>
<td>4096</td>
</tr>
</tbody>
</table>

Some of these members apply to other types of controls that have owner-draw facilities.

Normally, the \textit{BackColor} property of the \textit{DrawItemEventArgs} object will be \textit{SystemColors.Window} and the \textit{ForeColor} property will be \textit{SystemColors.WindowText}. To be consistent with normal menu items, these are not the colors you should be using! Use \textit{SystemColors.Menu} and \textit{SystemColors.MenuText} instead. If the item is selected, \textit{BackColor} will be \textit{SystemColors.Highlight} and \textit{ForeColor} will be \textit{SystemColors.HighlightText}. These are the correct colors for selected menu items.

The \textit{Font} property of the \textit{DrawItemEventArgs} property will be the same as \textit{SystemInformation.MenuFont}.

In addition, \textit{DrawItemEventArgs} has two methods that assist you in drawing the item:

\textbf{\textit{DrawItemEventArgs} Methods}

\begin{verbatim}
    void DrawBackground();
    void DrawFocusRectangle();
\end{verbatim}

The \textit{DrawFocusRectangle} method isn't used with menu items.

You'll also find the following static method of the \textit{ControlPaint} class to be useful for drawing arrows, check marks, and radio buttons on menus:

\textbf{\textit{ControlPaint Static DrawMenuGlyph Methods}}

\begin{verbatim}
    void DrawMenuGlyph(Graphics grfx, Rectangle rect, MenuGlyph mg)
    void DrawMenuGlyph(Graphics grfx, int x, int y, int cx, int cy, MenuGlyph mg)
\end{verbatim}
MenuGlyph is another enumeration:

**MenuGlyph Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
</tr>
<tr>
<td>Arrow</td>
<td>0</td>
</tr>
<tr>
<td>Checkmark</td>
<td>1</td>
</tr>
<tr>
<td>Bullet</td>
<td>2</td>
</tr>
<tr>
<td>Max</td>
<td>2</td>
</tr>
</tbody>
</table>

You can size your menu items in a couple ways. The normal font used for menu items is (as I mentioned) available from `SystemInformation.MenuFont`. Another important measure is `SystemInformation.MenuCheckSize`, which is the default width and height of the check mark. As you can see in the static `ControlPaint.DrawMenuGlyph` method, you specify the width and height of the glyph (such as the check mark) as you draw it. If you make your items taller than the normal menu items and you want to use check marks, you should probably scale the check mark glyph when you draw it. This implies that you should also take account of the scaled-up width of the check mark when you calculate the size of the item while handling the `MeasureItem` event.

Here's a program that has a single top-level item named Facename. The pop-up menu has three items showing the three most common font faces. The owner-draw logic displays these facename items in fonts based on the facenames.

**OwnerDrawMenu.cs**

```csharp
// OwnerDrawMenu.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Text;    // For HotkeyPrefix enumeration
using System.Windows.Forms;

class OwnerDrawMenu: Form
{
    const int iFontSize = 18; // For menu items
    MenuItem miFacename;

    public static void Main()
    {
        Application.Run(new OwnerDrawMenu());
    }
    public OwnerDrawMenu()
    {
        Text = "Owner-Draw Menu";

        // Top-level items
```
Menu = new MainMenu();
Menu.MenuItems.Add("&Facename");

// Array of items on submenu

string[] astrText = {"&Times New Roman", "&Arial", "&Courier New");
MenuItem [] ami = new MenuItem[astrText.Length];

EventHandler ehOnClick = new EventHandler(MenuFacenameOnClick);
MeasureItemEventHandler ehOnMeasureItem =
    new MeasureItemEventHandler(MenuFacenameOnMeasureItem);
DrawItemEventHandler ehOnDrawItem =
    new DrawItemEventHandler(MenuFacenameOnDrawItem);

for (int i = 0; i < ami.Length; i++)
{
    ami[i] = new MenuItem(astrText[i]);
    ami[i].OwnerDraw = true;
    ami[i].RadioCheck = true;
    ami[i].Click += ehOnClick;
    ami[i].MeasureItem += ehOnMeasureItem;
    ami[i].DrawItem += ehOnDrawItem;
}

miFacename = ami[0];
miFacename.Checked = true;

Menu.MenuItems[0].MenuItems.AddRange(ami);
}
void MenuFacenameOnClick(object obj, EventArgs ea)
{
    miFacename.Checked = false;
    miFacename = (MenuItem) obj;
    miFacename.Checked = true;

    Invalidate();
}
void MenuFacenameOnMeasureItem(object obj, MeasureItemEventArgs miea)
{
    MenuItem mi = (MenuItem) obj;
    Font font = new Font(mi.Text.Substring(1), iFontPointSize);

    StringFormat strfmt = new StringFormat();
    strfmt.HotkeyPrefix = HotkeyPrefix.Show;
SizeF sizeof = miea.Graphics.MeasureString(mi.Text, font, 1000, strfmt);

miea.ItemWidth  = (int) Math.Ceiling(sizeof.Width);
miea.ItemHeight = (int) Math.Ceiling(sizeof.Height);

miea.ItemWidth += SystemInformation.MenuCheckSize.Width *
    miea.ItemHeight / SystemInformation.MenuCheckSize.Height;

miea.ItemWidth -= SystemInformation.MenuCheckSize.Width;
}

void MenuFacenameOnDrawItem(object obj, DrawItemEventArgs diea)
{
    MenuItem mi   = (MenuItem) obj;
    Graphics grfx = diea.Graphics;
    Brush brush;

    // Create the Font and StringFormat.
    Font font = new Font(mi.Text.Substring(1), iFontPointSize);
    StringFormat strfmt = new StringFormat();
    strfmt.HotkeyPrefix = HotkeyPrefix.Show;

    // Calculate check mark and text rectangles.
    Rectangle rectCheck = diea.Bounds;
    rectCheck.Width = SystemInformation.MenuCheckSize.Width *
        rectCheck.Height / SystemInformation.MenuCheckSize.Height;

    Rectangle rectText = diea.Bounds;

    rectText.X += rectCheck.Width;

    // Do all the drawing.
    diea.DrawBackground();

    if ((diea.State & DrawItemState.Checked) != 0)
        ControlPaint.DrawMenuGlyph(grfx, rectCheck, MenuGlyph.Bullet);

    if ((diea.State & DrawItemState.Selected) != 0)
        brush = SystemBrushes.HighlightText;
else
    brush = SystemBrushes.FromSystemColor(SystemColors.MenuText);
    grfx.DrawString(mi.Text, font, brush, rectText, strfmt);
}
protected override void OnPaint(PaintEventArgs pea)
{
  Graphics grfx = pea.Graphics;
  Font font = new Font(miFacename.Text.Substring(1), 12);
  StringFormat strfmt = new StringFormat();
  strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
  grfx.DrawString(Text, font, new SolidBrush(ForeColor), 0, 0);
}

I've set the _iFontPointSize_ field to 18 just to have a jumbo font in the menu to ensure that the measuring and drawing logic is working correctly.

The _MenuFacenameOnMeasureItem_ method begins by obtaining the _MenuItem_ to be measured and constructing a font based on the _Text_ property of that item:

.MenuItem mi = (MenuItem) obj;
Font font = new Font(mi.Text.Substring(1), iFontPointSize);

The _Substring_ method on the _Text_ property skips past the ampersand. Next, the method creates a _StringFormat_ object indicating that the letter following the ampersand will be underlined when the facename is displayed:

.StringFormat strfmt = new StringFormat();
strfmt.HotkeyPrefix = HotkeyPrefix.Show;

The _Text_ property of the menu item is then measured based on the new _Font_ and _StringFormat_ objects:

.SizeF sizef = miea.Graphics.MeasureString(mi.Text, font, 1000, strfmt);

Without a check mark, the _sizef_ structure would provide the size of the menu item:

.miea.ItemWidth  = (int) Math.Ceiling(sizef.Width);
.miea.ItemHeight = (int) Math.Ceiling(sizef.Height);

But the width must be increased by the width of the check mark when the height of the check mark is scaled to the height of the text

.miea.ItemWidth += SystemInformation.MenuCheckSize.Width *
    miea.ItemHeight / SystemInformation.MenuCheckSize.Height;

and then decreased by the normal width of the check mark:

.miea.ItemWidth -= SystemInformation.MenuCheckSize.Width;

The _MenuFacenameOnDrawItem_ method creates _Font_ and _StringFormat_ objects similarly and then calculates two _Rectangle_ structures based on the _Bounds_ property of the _DrawItemEventArgs_ object. The first rectangle is the location and size of the check mark:

.Rectangle rectCheck = diea.Bounds;
rectCheck.Width = SystemInformation.MenuCheckSize.Width *
    rectCheck.Height /
    SystemInformation.MenuCheckSize.Height;

The second is the location and size of the text string:
Rectangle rectText = diea.Bounds;
rectText.X += rectCheck.Width;

From that point, it's simple. The DrawBackground method draws the background, DrawMenuGlyph draws the check mark, and DrawString draws the text, the color of which is based on whether or not the item is selected. And here's the result:

For some simple applications, such extensive processing of the MeasureItem and DrawItem events isn't required. For example, the following program loads a 64-pixel-square bitmap resource and uses this image as a menu item.

HelpMenu.cs
//---------------------------------------
// HelpMenu.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelpMenu : Form
{
    Bitmap bmHelp;

    public static void Main()
    {
        Application.Run(new HelpMenu());
    }
    public HelpMenu()
    {
        Text = "Help Menu";
    }
bmHelp = new Bitmap(GetType(), "HelpMenu.Bighelp.bmp");

Menu = new MainMenu();
Menu.MenuItems.Add("&Help");

MenuItem mi = new MenuItem("&Help");
mi.OwnerDraw = true;
mi.Click += new EventHandler(MenuHelpOnClick);
mi.DrawItem += new DrawItemEventHandler(MenuHelpOnDrawItem);
mi.MeasureItem +=
    new MeasureItemEventHandler(MenuHelpOnMeasureItem);

Menu.MenuItems[0].MenuItems.Add(mi);
}
void MenuHelpOnMeasureItem(object obj, MeasureItemEventArgs miea)
{
    miea.ItemWidth = bmHelp.Width;
    miea.ItemHeight = bmHelp.Height;
}
void MenuHelpOnDrawItem(object obj, DrawItemEventArgs diea)
{
    Rectangle rect = diea.Bounds;
    rect.Width = bmHelp.Width;

    diea.DrawBackground();
    diea.Graphics.DrawImage(bmHelp, rect);
}
void MenuHelpOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Help not yet implemented.", Text);
}

Bighelp.bmp
The `MeasureItem` and `DrawItem` processing here is very modest. The `MeasureItem` handler needs only set `ItemWidth` and `ItemHeight` to the height and width of the bitmap, and `DrawItem` draws it, essentially right-justifying the image within the rectangle indicated by the `Bounds` property. The resulting effect perhaps mirrors the desperation of a new user:
Chapter 15: Paths, Regions, and Clipping

Overview

If you’ve ever done any graphics programming in PostScript, you probably already know what a graphics path is. In PostScript, you can’t get to first base without using paths. While other graphics programming environments haven’t gone to quite the extremes of PostScript in elevating the path to the role of central drawing object, the path has come to be recognized as a valuable graphics programming tool.

Very simply, the graphics path provides a way to connect straight lines and curves. As you know, you can draw connected straight lines using DrawLines and connected Bézier curves using DrawBeziers, but I haven’t yet discussed any way to connect straight lines and Bézier curves to each other. That’s what the path does. It sounds simple, but it opens up a variety of drawing techniques that I’ll explore in this chapter and in Chapters 17 and 19.

You can also use paths for clipping. Clipping is the restriction of graphics output to a particular area of the screen or printer page. When you specify a path for clipping, the path is actually converted to a region first. A region describes an area of the output device in device coordinates.

A Problem and Its Solution

Let’s begin our exploration of graphics paths with a graphics programming problem. Suppose you want to draw a figure that’s composed of a line, a semicircle, and another line, all connected to each other, and you want to use a pen that is considerably thicker than 1 pixel. Here’s a possible first stab at drawing such a figure.

LineArcCombo.cs

```csharp
//-------------------------------------------
// LineArcCombo.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class LineArcCombo : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new LineArcCombo());
    }
    public LineArcCombo()
    {
        Text = "Line & Arc Combo";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr, 25);
        grfx.DrawLine(pen, 25, 100, 125, 100);
        grfx.DrawArc(pen, 125, 50, 100, 100, -180, 180);
        grfx.DrawLine(pen, 225, 100, 325, 100);
    }
}
```
The two lines are 100 units in length (that's 100 pixels on the video display and 1 inch on the printer), and the circle that forms the basis of the arc is 100 units in diameter. The pen is 25 units wide. And the result looks like this:

![Image](image1.png)

Perhaps this is exactly what you wanted. But it's not what I wanted. I wanted the lines and arc to be connected. Sure, they're touching each other, but they are definitely not visually connected. I don't want those notches on the bottom inside of the arc.

If you alter the LineArcCombo program to draw the figure twice, once with a thick gray pen and then with a 1-pixel-wide black pen, you might more clearly see what's going on here:

![Image](image2.png)

The 25-pixel-wide lines simply extend 12 pixels to each side of the 1-pixel-wide lines. Because the lines and arc are drawn with separate method calls, each figure is a distinct entity. At the two points at which the lines and arc meet, the wide lines intersect but do not form a composite whole.

You could perhaps finagle the coordinates to make this figure look right. You could, for example, lower the arc by 12 units or so. But deep in your heart of hearts, you know that you haven't solved the problem, only temporarily disguised it.

What we need here is some way of letting the graphics system know that the lines and arc are supposed to be connected. If you were dealing only with straight lines, drawing connected lines would be a snap: you'd use `DrawLines` to draw a polyline rather than `DrawLine` to draw discrete lines. For example, here's a program that draws something similar to what we want.

```
WidePolyline.cs
//-------------------------------------------
// WidePolyline.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class WidePolyline : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new WidePolyline());
    }
}
```

public WidePolyline()
{
    Text = "Wide Polyline";
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Pen pen = new Pen(clr, 25);
    grfx.DrawLines(pen, new Point[] {
        new Point(25, 100), new Point(125, 100),
        new Point(125, 50), new Point(225, 50),
        new Point(225, 100), new Point(325, 100) });
}

The **DrawLines** call includes an array of six **Point** structures to render a polyline that's composed of five lines:

![Wide Polyline](image)

The graphics system knows that these lines are supposed to be connected because they're all included in one function call. The wide line is correctly drawn where the lines meet.

The use of a polyline in the WidePolyline program suggests another solution to the line-and-arc figure. You could look back in *Chapter 5* to see how to draw an ellipse using a polyline, and then implement the arc in that way. Or you could convert the straight lines to Bézier splines (by specifying control points that are between and collinear to the end points), convert the arc to one or more Bézier splines (using formulas shown in *Chapter 13*), and then draw the whole thing using **DrawBeziers**.

But surely there's a more direct approach to letting the graphics system know that lines and arc are connected. What we need is something like **DrawLines** that works with a combination of straight lines and arcs. And while we're at it, we may as well request that this magical function work with Bézier splines and cardinal splines as well.

That magical function (more precisely, a magical class) is **GraphicsPath**. Here's a program named LineArcPath that correctly draws the figure using only three more statements than LineArcCombo.

**LineArcPath.cs**

```csharp
//------------------------------------------
// LineArcPath.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
```
class LineArcPath: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new LineArcPath());
    }
    public LineArcPath()
    {
        Text = "Line & Arc in Path";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        Pen pen = new Pen(clr, 25);

        path.AddLine(25, 100, 125, 100);
        path.AddArc(125, 50, 100, 100, -180, 180);
        path.AddLine(225, 100, 325, 100);

        grfx.DrawPath(pen, path);
    }
}

One of these three additional statements creates the path at the beginning of the DoPage method:
GraphicsPath path = new GraphicsPath();

Although the class that implements the path is named GraphicsPath, I'll be using just the simple
variable name path for instances of this class. GraphicsPath is defined in the
System.Drawing.Drawing2D namespace, and an additional using statement accounts for another of
the three additional statements in this program.

The LineArcCombo program drew the first line using the DrawLine method of the Graphics class:
grfx.DrawLine(pen, 25, 100, 125, 100);

The LineArcPath program replaces this statement with the AddLine method of the GraphicsPath
class:
path.AddLine(25, 100, 125, 100);

The AddLine method doesn't have a Pen argument, but otherwise the arguments are identical to
DrawLine. The same is true for AddArc as compared to DrawArc. The AddLine and AddArc calls
don't draw anything. The coordinates specified in the method calls are simply accumulated in the
path.

Finally (in the third of the three additional statements in this program), the path is actually drawn on
the display device:
grfx.DrawPath(pen, path);

Notice that DrawPath is a method of our old friend the Graphics class. The result of the DrawPath
call is exactly the figure we were hoping for:
Programmers with experience in Win32 API or MFC programming will notice that the implementation of the graphics path in Windows Forms is conceptually different. In the Win32 API, the `BeginPath` function puts the device context into a special mode where all calls to the normal drawing functions (`LineTo`, `BezierTo`, and so forth) are not rendered but instead become part of the path. The path is ended with a call to `EndPath` and then drawn with a call to `StrokePath` (or something else is done with the path).

The Windows Forms approach is much more flexible. The Win32 API allows only one path to be in existence for a particular device context, but with Windows Forms, you can create and store as many paths as you want. And you don't need a `Graphics` object to create a path. The path exists independently of any `Graphics` object until it is rendered using `DrawPath` (or you do something else with the path).

You could, in fact, alter LineArcPath to store the `GraphicsPath` object as a field. You could create the path and make the calls to `AddLine` and `AddArc` in the form's constructor. The `DoPage` method would then just create the pen and call `DrawPath`. If you really wanted to get everything out of the way in the form's constructor, you could also make the pen-creation statement a field of the form.

### The Path, More Formally

Let's begin a more formal look at paths with a few definitions and a quick overview:

A *path* is a collection of device-independent coordinate points that describe straight lines and curves. These lines and curves might or might not be connected to each other. Any set of connected lines and curves within the path is known as a *figure* or a *subpath*. (Both terms are used in the Windows Forms interface. The terms are synonymous.) Thus, a path is composed of zero or more subpaths. Each subpath is a collection of connected lines and curves. The path created in the LineArcPath program has just one subpath.

A subpath can be either open or closed. A subpath is closed if the end of the last line in the subpath is connected to the beginning of the first line. (A special method in the `GraphicsPath` class—`CloseFigure`—is available to close a subpath.) Otherwise, the subpath is open. The single subpath created in the LineArcPath program is open.

I've already demonstrated the `DrawPath` method that draws the lines and curves that comprise a path on an output device. The `Graphics` class also includes a `FillPath` method that uses a brush to fill the interior of all closed subpaths in the path. For the purpose of the `FillPath` method, all open subpaths in the path are closed so that all the subpaths define enclosed areas.

As I'll demonstrate later in this chapter, you can also convert a path into a *region*. In contrast to a path (which is a collection of lines and curves), a region describes an area of the display surface. This area may be simple (a rectangle) or quite complex. You can fill the area defined by a region with a brush. Or you can use the region for clipping. Clipping restricts drawing to a particular area of the display surface.

Programmers approaching paths for the first time sometimes tend to think that a path is something much more than just a collection of lines and curve definitions. Let's disabuse ourselves of this notion by looking at the `GraphicsPath` properties. The path contains no other persistent data than what's accessible through its properties:

<table>
<thead>
<tr>
<th><code>GraphicsPath</code> Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><code>FillMode</code></td>
</tr>
<tr>
<td><code>int</code></td>
</tr>
</tbody>
</table>
The FillMode enumeration is also used with the DrawPolygon method described in Chapter 5 and the DrawClosedCurve method featured in Chapter 13. For paths, the FillMode property determines how the path is filled (or how it’s converted to a region) when the path contains overlapping lines.

The other four properties redundantly define nothing more than two arrays of identical size:
- An array of PointF structures named PathPoints
- An array of byte values named PathTypes

The number of elements in these arrays (which you could obtain from PathPoints.Length or PathTypes.Length) is also provided by the PointCount property.

An additional layer of redundancy is provided by the PathData property. This property is an object of type PathData, defined in the System.Drawing.Drawing2D namespace. The PathData class has the following two properties:

### PathData Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointF[]</td>
<td>Points</td>
<td>get/set</td>
<td>Array of coordinate points</td>
</tr>
<tr>
<td>byte[]</td>
<td>Types</td>
<td>get/set</td>
<td>Array of point types</td>
</tr>
</tbody>
</table>

For any GraphicsPath object, the Points array of the PathData property is identical to the PathPoints property; the Types array of the PathData property is identical to the PathTypes property.

The values in the array of bytes that makes up the PathTypes property are actually values of the PathPointType enumeration, also defined in System.Drawing.Drawing2D:

### PathPointType Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0</td>
</tr>
<tr>
<td>Line</td>
<td>1</td>
</tr>
<tr>
<td>Bezier or Bezier3</td>
<td>3</td>
</tr>
<tr>
<td>PathTypeMask</td>
<td>7</td>
</tr>
<tr>
<td>DashMode</td>
<td>16</td>
</tr>
<tr>
<td>PathMarker</td>
<td>32</td>
</tr>
<tr>
<td>CloseSubpath</td>
<td>128</td>
</tr>
</tbody>
</table>

Each PointF structure in the PathPoints array has an associated PathPointType of Start, Line, or Bezier. The Start type identifies the first point of a figure; the Line type indicates a point that defines a straight line; the Bezier type indicates a point that is part of a Bézier spline. Any arcs or canonical splines are converted into Bézier splines as they are added to the path. After my demonstration in Chapter 13 about how circles can be closely approximated using Bézier splines, such conversions should be plausible.

The last three values in the PathPointType enumeration table are flags that can be combined with the values of Start, Line, or Bezier. As you’ll see, both the PathMarker and CloseSubpath flags are generated by GraphicsPath method calls.
The `PathTypeMask` enumeration member is a bit mask that lets you separate the values into point types (Start, Line, or Bezier) and flags (DashMode, PathMarker, or CloseSubpath).

What a path does not contain is anything that relates these coordinate points to real-world measurements. It is meaningless to ask if the coordinate points in a path are pixels or inches or millimeters or anything else. They're just points. They are converted to pixels, inches, or millimeters only when the path is rendered on an output device.

**Creating the Path**

The `GraphicsPath` class has six constructors:

<table>
<thead>
<tr>
<th><code>GraphicsPath Constructors</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphicsPath()</td>
</tr>
<tr>
<td>GraphicsPath(Point[] apt, byte[] abyPointType)</td>
</tr>
<tr>
<td>GraphicsPath(PointF[] aptf, byte[] abyPointType)</td>
</tr>
<tr>
<td>GraphicsPath(FillMode fm)</td>
</tr>
<tr>
<td>GraphicsPath(Point[] apt, byte[] abyPointType, FillMode fm)</td>
</tr>
<tr>
<td>GraphicsPath(PointF[] aptf, byte[] abyPointType, FillMode fm)</td>
</tr>
</tbody>
</table>

If you don't specify a `FillMode` argument, the default is `FillMode.Alternate`.

As four of the constructors indicate, you can create a path using an array of `Point` or `PointF` structures and an array of corresponding `PathPointType` enumeration values, expressed as an array of `byte` values. But it's unlikely that a program will start off creating a path in that way. Instead, these constructors are most profitably used to alter the `PathPoints` values of an existing path.

Normally, you create a new path using the default constructor:

```csharp
GraphicsPath path = new GraphicsPath();
```

You then call methods of the `GraphicsPath` class that add straight lines and curves to the path. These methods are similar to corresponding methods in the `Graphics` class except that they begin with the word `Add` instead of `Draw` and they have no `Pen` argument.

Here are the methods of `GraphicsPath` that add straight lines, Bézier splines, arcs, and canonical splines to the current subpath. I'm not showing the arguments in the following table because for the most part they are the same as corresponding `Draw` methods defined in the `Graphics` class:

<table>
<thead>
<tr>
<th><code>GraphicsPath Methods (selection)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>void AddLine(...)</td>
</tr>
<tr>
<td>void AddLines(...)</td>
</tr>
<tr>
<td>void AddArc(...)</td>
</tr>
<tr>
<td>void AddBezier(...)</td>
</tr>
<tr>
<td>void AddBeziers(...)</td>
</tr>
<tr>
<td>void AddCurve(...)</td>
</tr>
</tbody>
</table>

Arcs and canonical splines are converted to Bézier splines as they are added to the path.

If `path` is an object of type `GraphicsPath`, the following three calls add three connected lines to the path:

```csharp
path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
```
path.AddLine(100, 100, 100, 0);

The resultant figure looks like the left, bottom, and right sides of a square. I chose the coordinates so that the end point of each line is the same as the starting point of the next line, just as if I were drawing such a figure.

When defining a path, however, it's not necessary to be quite this meticulous. Until you specify otherwise (as I'll demonstrate shortly), the lines, arcs, Bézier splines, and canonical splines you add to the path all end up being part of the same figure. If the coordinates don't match up exactly, the path automatically generates a straight line to connect the pieces. You can achieve the same results as the three statements just shown by eliminating the second statement entirely:

path.AddLine(0, 0, 100);
path.AddLine(100, 100, 100, 0);

Because the first line ends at (0, 100) and the second line begins at (100, 100), the path adds a line between those two points.

You can also make calls to the following three methods:

**GraphicsPath Methods (selection)**

void StartFigure()
void CloseFigure()
void CloseAllFigures()

All three of these calls end the current subpath and begin a new subpath. In addition, CloseFigure closes the current subpath. If necessary, a straight line is automatically added to the path from the last point of the subpath to the first point of the subpath. CloseAllFigures closes all the subpaths that are part of the path so far.

The calls

path.AddLine(0, 0, 0, 100);
path.AddLine(100, 100, 100, 0);
path.AddLine(0, 0, 100, 100);
path.AddLine(100, 100, 100, 0);
path.AddLine(0, 0, 0, 100);
path.CloseFigure();

explicitly create a square closed figure. The calls

path.AddLine(0, 0, 0, 100);
path.AddLine(100, 100, 100, 0);
path.CloseFigure();

create the same closed figure by forcing the path to automatically add lines for the bottom and top sides. The calls

path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100, 100, 0);
path.AddLine(100, 0, 0, 0);
path.StartFigure();

create a figure that consists of four sides of a square, but the figure isn't considered closed because it doesn't end with a call to CloseFigure.

The following methods start a new figure, which is then closed:
For example, the calls

```csharp
path.AddLine(0, 0, 100, 0);
path.AddRectangle(new Rectangle(50, 50, 100, 100));
path.AddLine(200, 0, 0, 0);
```

create three subpaths:
- One line, unclosed
- Four lines, closed
- One line, unclosed

You can also add one path to another path:

**GraphicsPath AddPath Method**

```csharp
void AddPath(GraphicsPath path, bool bConnect)
```

The second argument indicates whether the path that is added should be connected to the current subpath.

The `AddString` methods add a text string to the path. The syntax of these methods is quite different from the syntax of the `DrawString` methods:

**GraphicsPath AddString Methods**

```csharp
void AddString(string str, FontFamily ff, int iStyle, float fSize, Point pt, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize, PointF ptf, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize, Rectangle rect, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize, RectangleF rectf, StringFormat sf)
```

Despite the presence of arguments that don't look a bit like coordinate points, these methods do nothing more than add a series of straight lines and Bézier curves to the path. The lines and curves are the outlines of the font characters.

The arguments to `AddString` are actually not as odd as the method definitions initially suggest. The third argument is defined as an `int` but is really a member of the `FontStyle` enumeration (`Regular`, `Bold`, `Italic`, `Underline`, or `Strikeout`). The second, third, and fourth arguments are thus the same as three arguments used in a constructor to `Font`. 
But why don’t the AddString methods use Font arguments in the same way that DrawString does? Because a Font is most commonly a specific point size, and a path doesn’t retain any metrical information. The fSize argument to AddString is not a point size. Specifying the fSize argument to AddString is similar to creating a Font with a pixel size and an argument of GraphicsUnit.Pixel or GraphicsUnit.World, as I discussed in Chapter 9 (on page 379). The text doesn’t assume a metrical size until it’s rendered.

Putting text into a path opens up such a wide variety of effects that Chapter 19 is entitled "Font Fun."

You can also insert nonfunctional markers into the path:

**GraphicsPath Methods (selection)**

```csharp
void SetMarkers()
void ClearMarkers()
```

You can then use the GraphicsPathIterator class to search for these markers. Such a facility possibly lets you edit a path with more ease.

**Rendering the Path**

Most often, you call one of the following two methods of the Graphics class to render a path:

**Graphics Methods (selection)**

```csharp
void DrawPath(Pen pen, GraphicsPath path)
void FillPath(Brush brush, GraphicsPath path)
```

The DrawPath method draws the lines and curves that comprise the path using the specified pen. FillPath fills the interiors of all closed subpaths using the specified brush. The method closes all unclosed subpaths for purposes of this function but doesn’t permanently affect the path. If any lines of the path overlap, interiors are filled based on the current FillPath property of the GraphicsPath object. At the time of rendering, the points in the path are subject to any transforms that are in effect in the Graphics object.

Let’s see how this stuff works in practice. The Flower program draws a flower using a path and a transform.

**Flower.cs**

```csharp
// Flower.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Flower: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new Flower());
    }
}```
public Flower()
{
    Text = "Flower";
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    // Draw green stem from lower left corner to center.
    grfx.DrawBezier(new Pen(Color.Green, 10),
                    new Point(0, cy),
                    new Point(0, 3 * cy / 4),
                    new Point(cx / 4, cy / 4),
                    new Point(cx / 2, cy / 2));

    // Set up transform for remainder of flower.
    float fScale = Math.Min(cx, cy) / 2000f;
    grfx.TranslateTransform(cx / 2, cy / 2);
    grfx.ScaleTransform(fScale, fScale);

    // Draw red petals.
    GraphicsPath path = new GraphicsPath();
    path.AddBezier(new Point(0, 0), new Point(125, 125),
                   new Point(475, 125), new Point(600, 0));
    path.AddBezier(new Point(600, 0), new Point(475, -125),
                   new Point(125, -125), new Point(0, 0));
    for (int i = 0; i < 8; i++)
    {
        grfx.FillPath(Brushes.Red, path);
        grfx.DrawPath(Pens.Black, path);
        grfx.RotateTransform(360 / 8);
    }

    // Draw yellow circle in center.
    Rectangle rect = new Rectangle(-150, -150, 300, 300);
    grfx.FillEllipse(Brushes.Yellow, rect);
    grfx.DrawEllipse(Pens.Black, rect);
}

The DoPage method begins by drawing a Bézier spline from the lower left corner of the client area (or printer page) to the center to create the stem. Next, it sets up a world transform that creates a four-quadrant isotropic drawing area with the origin in the center and coordinates ranging from −1000 to 1000.
The program needs to draw some petals next, and that's where the path comes into play. If petals looked like ellipses, I could just use *FillEllipse*. But petals are more accurately defined with a pair of Bézier splines, and filling such a figure requires a path. After the program creates the path, it calls *FillPath* and *DrawPath* eight times. After each pair of calls, the *RotateTransform* call changes the world transform of the *Graphics* object so that the eight petals are rotated around the center. *DoPage* finishes by drawing a yellow circle in the center of the client area.

I'm sure you remember the Scribble program from Chapter 8. At the time, I demonstrated how to save all the lines the user draws using the *ArrayList* class, which is an array-like object that can dynamically resize itself. The use of the *ArrayList* class is actually quite similar to saving coordinates in a path. And using a *GraphicsPath* object instead of an *ArrayList* object simplifies the program considerably. It's even simpler than the version in Chapter 11 (ScribbleWithBitmap) that saved the image using a shadow bitmap.

*ScribbleWithPath.cs*

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ScribbleWithPath: Form
{
    GraphicsPath path;
    bool bTracking;
    Point ptLast;

    public static void Main()
    {
        Application.Run(new ScribbleWithPath());
    }

    public ScribbleWithPath()
    {
```
Text = "Scribble with Path";
BackColor = SystemColors.Window;
ForeColor = SystemColors.WindowText;

    // Create the path.

    path = new GraphicsPath();
}
protected override void OnMouseDown(MouseEventArgs mea)
{
    if (mea.Button != MouseButtons.Left)
        return;

    ptLast = new Point(mea.X, mea.Y);
    bTracking = true;

    // Start a figure.

    path.StartFigure();
}
protected override void OnMouseMove(MouseEventArgs mea)
{
    if (!bTracking)
        return;

    Point ptNew = new Point(mea.X, mea.Y);
    bTracking = true;

    // Start a figure.

    path.StartFigure();
}
protected override void OnMouseUp(MouseEventArgs mea)
{
    bTracking = false;
}
protected override void OnPaint(PaintEventArgs pea)
{
    // Draw the path.

    pea.Graphics.DrawPath(new Pen(ForeColor), path);
Aside from the additional using statement, transforming the no-save version of Scribble to ScribbleWithPath requires defining a path as a field variable and then adding just four statements, all identified with comments.

The path is created in the form's constructor. Whenever the left mouse button is pressed when the cursor is positioned in the form's client area, a call to the StartFigure method begins a new subpath. An AddLine call during the OnMouseMove method adds a new line to the path. The OnPaint method is simply a call to DrawPath.

### Path Transforms

The GraphicsPath class contains several methods that let a program modify a path. The first of these is likely to be very confusing. (At least it was for me when I first encountered it.)

#### GraphicsPath Transform Method

```csharp
void Transform(Matrix matrix)
```

As you know from Chapter 7, the Graphics class has a property named `Transform` that is of type `Matrix`. The `Transform` property of the Graphics class affects the display of all subsequent graphics output.

But the `Transform` in GraphicsPath is different. `Transform` is not a property of GraphicsPath; `Transform` is a method. And that's an important distinction. A property is usually a characteristic of an object; a method usually carries out an operation. A property is an adjective; a method is a verb.

The `Transform` method of the GraphicsPath class permanently alters the coordinates of the path by applying the specified transform to those coordinates. The `Transform` method doesn't affect coordinates subsequently added to the path. Nor does the GraphicsPath object retain the transform in any way. For example, if you have a Matrix object named `matrix` that describes a doubling of coordinate points, and you call

```csharp
gfx.Transform(matrix);
```

the result is equivalent to obtaining the array of coordinate points in the path using the `PathPoints` property, doubling all the numbers in the array, and then creating a new path based on those modified points.

The `Transform` method is the only method in the GraphicsPath class concerned with matrix transforms. To use it, you'll need to make use of the Matrix class, which is defined in the System.Drawing.Drawing2D namespace and which I touched on briefly toward the end of Chapter 7. The easiest way to use the Matrix class is first to create an identity matrix using the default constructor:

```csharp
Matrix matrix = new Matrix();
```

You can then use various methods of the Matrix class to alter this transform. The `Translate` method is just like the TranslateTransform method of the Graphics class. (In fact, I wouldn't be surprised if the Graphics class implemented its TranslateTransform methods by simply calling the corresponding Translate method of its Transform property.)

#### Matrix Translate Methods

```csharp
void Translate(float dx, float dy)
void Translate(float dx, float dy, MatrixOrder mo)
```

The MatrixOrder enumeration has two members, Append and Prepend.
The *Scale* method is just like the *ScaleTransform* method of the *Graphics* class:

**Matrix Scale Methods**

```csharp
void Scale(float sx, float sy)
void Scale(float sx, float sy, MatrixOrder mo)
```

Earlier I mentioned doubling all the coordinates in a path. You can do that with the following three lines of code:

```csharp
Matrix matrix = new Matrix();
matrix.Scale(2, 2);
path.Transform(matrix);
```

The *Matrix* class also includes a *Rotate* method:

**Matrix Rotate Methods**

```csharp
void Rotate(float fAngle)
void Rotate(float fAngle, MatrixOrder mo)
```

You can alter the Flower program to use the *Rotate* method of the *Matrix* class rather than the *RotateTransform* method of the *Graphics* class. After creating the path, create a *Matrix* object that describes a rotation of 45 degrees:

```csharp
Matrix matrix = new Matrix();
matrix.Rotate(45);
```

Then in the `for` loop, rather than call *RotateTransform*, call the *Transform* method of the path:

```csharp
path.Transform(matrix);
```

In the original version of Flower, the path remains the same and the *RotateTransform* call affects how the coordinates are transformed as the path is rendered by the *Graphics* class. In the altered version, the coordinates stored in the path are rotated. By the end of the `for` loop, after eight rotations of 45 degrees, the coordinates of the path have been restored to their original values.

Here's an interesting method of the *Matrix* class that isn't duplicated by a method in the *Graphics* class:

**Matrix RotateAt Methods**

```csharp
void RotateAt(float fAngle, PointF ptf)
void RotateAt(float fAngle, PointF ptf, MatrixOrder mo)
```

Normally the matrix transform rotates an image around the point (0, 0). This method lets you specify the point around which the rotation occurs. For example, suppose you create a path like this:

```csharp
GraphicsPath path = new GraphicsPath();
path.AddRectangle(new Rectangle(0, 0, 100, 100));
```

The path contains the points (0, 0), (100, 0), (100, 100), and (0, 100). If you then create a *Matrix* object, call the *Rotate* method for 45 degrees, and apply it to the path like so:

```csharp
Matrix matrix = new Matrix();
matrix.Rotate(45);
```
path.Transform(matrix);

the points in the path are, with some rounding, (0, 0), (70.7, 70.7), (0, 141.4), and (−70, 70). If instead you use the \texttt{RotateAt} method specifying the center of the rectangle

\begin{verbatim}
Matrix matrix = new Matrix();
matrix.RotateAt(45, new Point(50, 50));
path.Transform(matrix);
\end{verbatim}

the path contains the points (50, −20.7), (120.7, 50), (50, 120.7), and (−20.7, 50).

The \texttt{Matrix} class also includes a method for shearing:

\begin{verbatim}
Matrix Shear
void Shear(float xShear, float yShear)
void Shear(float xShear, float yShear, MatrixOrder mo)
\end{verbatim}

Applied to a default transform, this method results in the following transformation formulas:

\[
x' = x + xShear \cdot y \\
y' = yShear \cdot x + y
\]

\textbf{Other Path Modifications}

\texttt{Transform} is not the only method of the \texttt{GraphicsPath} class that modifies all the coordinates of a path. The \texttt{Flatten} method is intended to convert all the Bézier splines in a path into straight-line segments:

\begin{verbatim}
GraphicsPath Flatten
void Flatten()
void Flatten(Matrix matrix)
void Flatten(Matrix matrix, float fFlatness)
\end{verbatim}

You can optionally apply a \texttt{Matrix} to transform the points before flattening them.

The number of line segments decreases as the \texttt{fFlatness} argument gets higher. The default \texttt{fFlatness} argument is equivalent to an argument of 0.25. The method is not defined for values of 0.

The \texttt{Widen} method has a much more profound effect on the path than the \texttt{Flatten} method does. The first argument is always a \texttt{Pen} object:

\begin{verbatim}
GraphicsPath Widen
void Widen(Pen pen)
void Widen(Pen pen, Matrix matrix)
void Widen(Pen pen, Matrix matrix, float fFlatness)
\end{verbatim}

The method ignores the color of the pen and uses only the pen width, generally a width of at least a couple units or so. Imagine the path being drawn using a thick pen. The new path is the outline of that thick line. Every open path is converted into a closed path, and every closed path is converted into two closed paths. Before widening the path, the method converts all Bézier splines to polylines.
You can optionally specify a flatness factor for this conversion; you can also optionally use a Matrix to transform the coordinates in the path before the widening process.

The results of the Widen method are sometimes a bit strange, so it helps to look at an example. The following program creates a path in the constructor containing one open subpath shaped like a V, and one closed subpath shaped like a triangle.

WidenPath.cs

```csharp
// WidenPath.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WidenPath: PrintableForm
{
    GraphicsPath path;

    public new static void Main()
    {
        Application.Run(new WidenPath());
    }

    public WidenPath()
    {
        Text = "Widen Path";

        path = new GraphicsPath();

        // Create open subpath.
        path.AddLines(new Point[] {
            new Point(20, 10),
            new Point(50, 50),
            new Point(80, 10)
        });

        // Create closed subpath.
        path.AddPolygon(new Point[] {
            new Point(20, 30),
            new Point(50, 70),
            new Point(80, 30)
        });
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.ScaleTransform(cx / 300f, cy / 200f);

        for (int i = 0; i < 6; i++)
        {
```
{  
  GraphicsPath pathClone = (GraphicsPath) path.Clone();  
  Matrix matrix = new Matrix();  
  Pen penThin = new Pen(clr, 1);  
  Pen penThick = new Pen(clr, 5);  
  Pen penWiden = new Pen(clr, 7.5f);  
  Brush brush = new SolidBrush(clr);  
  
  matrix.Translate((i % 3) * 100, (i / 3) * 100);  
  
  if (i < 3)  
    pathClone.Transform(matrix);  
  else  
    pathClone.Widen(penWiden, matrix);  
  
  switch (i % 3)  
  {  
    case 0:  grfx.DrawPath(penThin, pathClone); break;  
    case 1:  grfx.DrawPath(penThick, pathClone); break;  
    case 2:  grfx.FillPath(brush, pathClone); break;  
  }  
}

The DoPage method makes six copies of this path using the Clone method and uses the Transform method to position each copy in a particular area of the display. It then draws the path six different ways. The results look like this:

![Image showing the path drawn in different ways]

The top row shows the path drawn with a 1-unit-wide pen, drawn with a 5-unit-wide pen, and filled. The bottom row shows the same three drawing operations following a call to Widen using a 7.5-unit-wide pen.

The two renditions on the left side of the display show most clearly the effects of the Widen method. The open V-shaped subpath is converted into a closed subpath that outlines the path as if it had been drawn with a wide pen. The closed triangle subpath is converted into two paths, one on the outside and one on the inside of the line that would result from drawing the path with a wide pen. Of course, the little interior loops at the apexes look rather odd, but those are the results of the algorithm that the Widen method uses.
The two renditions in the center column look just like the ones in the left column except drawn with a thicker pen.

The filled path in the upper right corner has an unfilled interior area as a result of the default filling mode of the path, which is FillMode.Alternating. Change the fill mode to FillMode.Winding and all interior areas will be filled. The most interesting version is the figure in the lower right corner. That's the effect of FillPath on the widened path. It looks very much like DrawPath on the original path using a wide pen.

You can determine the smallest rectangle in which the path can fit by using the GetBounds method, either with or without taking into account the effect of a matrix transform and a wide pen:

**GraphicsPath GetBounds Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>RectangleF GetBounds()</td>
</tr>
<tr>
<td>RectangleF GetBounds(Matrix matrix)</td>
</tr>
<tr>
<td>RectangleF GetBounds(Matrix matrix, Pen pen)</td>
</tr>
</tbody>
</table>

Neither argument has any effect on the coordinates stored in the path. You should be aware that the calculated rectangle reflects the minimum and maximum x and y coordinates of all the points in the path. If the path contains Bézier splines, the rectangle reflects the coordinates of the control points, not the actual curve. To get a more accurate measurement of the figure, call Flatten before GetBounds.

In Chapter 7, I spoke of the matrix transform as being a linear transform. The linearity of the transform imposes certain restrictions on what you can do with the transform. Parallelograms will always be transformed into other parallelograms, for example.

The GraphicsPath class introduces another transform in the Warp method. Like the Transform method, the Warp method modifies all the coordinates of the path. But the Warp transform is nonlinear, the only nonlinear transform in GDI+.

To use Warp, you specify four source coordinates and four destination coordinates. The method maps the four source coordinates to the four corresponding destination coordinates. The source coordinates are specified as a RectangleF structure. Conveniently (but not necessarily), you can set the RectangleF argument to the RectangleF structure returned from GetBounds. The destination coordinates are specified as an array of PointF structures:

**GraphicsPath Warp Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Warp(PointF[] aptfDst, RectangleF rectfSrc)</td>
</tr>
<tr>
<td>void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix)</td>
</tr>
<tr>
<td>void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix, WarpMode wm)</td>
</tr>
<tr>
<td>void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix, WarpMode wm, float fFlatness)</td>
</tr>
</tbody>
</table>

You can optionally also supply a Matrix object and a flatness value. The source points are transformed to the destination points like this:

- `aptfDst[0]` is the destination of the upper left corner of the rectangle.
- `aptfDst[1]` is the destination of the upper right corner of the rectangle.
- `aptfDst[2]` is the destination of the lower left corner of the rectangle.
- `aptfDst[3]` is the destination of the lower right corner of the rectangle.

An optional argument determines how intermediary points are calculated:
The PathWarping program lets you experiment with the Warp function. The form's constructor creates a path with a square 8-by-8 checkerboard pattern. You then use the mouse to indicate the destination of this path.

PathWarping.cs

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PathWarping : Form
{
    MenuItem miWarpMode;
    GraphicsPath path;
    PointF[] aptfDest = new PointF[4];

    public static void Main()
    {
        Application.Run(new PathWarping());
    }

    public PathWarping()
    {
        Text = "Path Warping";

        // Create menu.

        Menu = new MainMenu();

        Menu.MenuItems.Add("&Warp Mode");
        EventHandler ehClick = new EventHandler(MenuWarpModeOnClick);

        miWarpMode = new MenuItem("&" + (WarpMode)0, ehClick);
        miWarpMode.RadioCheck = true;
        miWarpMode.Checked = true;
        Menu.MenuItems[0].MenuItems.Add(miWarpMode);

        MenuItem mi = new MenuItem("&" + (WarpMode)1, ehClick);
        mi.RadioCheck = true;
```
Menu.MenuItems[0].MenuItems.Add(mi);

    // Create path.

    path = new GraphicsPath();

    for (int i = 0; i <= 8; i++)
    {
        path.StartFigure();
        path.AddLine(0, 100 * i, 800, 100 * i);
        path.StartFigure();

        path.AddLine(100 * i, 0, 100 * i, 800);
    }
    // Initialize Point array.

    aptfDest[0] = new Point(50, 50);
    aptfDest[1] = new Point(200, 50);
    aptfDest[2] = new Point(50, 200);
    aptfDest[3] = new Point(200, 200);

    void MenuWarpModeOnClick(object obj, EventArgs ea)
    {
        miWarpMode.Checked = false;
        miWarpMode = (MenuItem) obj;
        miWarpMode.Checked = true;

        Invalidate();
    }

    protected override void OnMouseDown(MouseEventArgs mea)
    {
        Point pt;

        if (mea.Button == MouseButtons.Left)
        {
            if (ModifierKeys == Keys.None)
                pt = Point.Round(aptfDest[0]);
            else if (ModifierKeys == Keys.Shift)
                pt = Point.Round(aptfDest[2]);
            else
                return;
        }
        else if (mea.Button == MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
pt = Point.Round(aptfDest[1]);
else if (ModifierKeys == Keys.Shift)
pt = Point.Round(aptfDest[3]);
else
    return;
}
else
    return;

Cursor.Position = PointToScreen(pt);
}
protected override void OnMouseMove(MouseEventArgs mea)
{
    Point pt = new Point(mea.X, mea.Y);

    if (mea.Button == MouseButtons.Left)
    {
        if (ModifierKeys == Keys.None)
            aptfDest[0] = pt;
        else if (ModifierKeys == Keys.Shift)
            aptfDest[2] = pt;
        else
            return;
    }
    else if (mea.Button == MouseButtons.Right)
    {
        if (ModifierKeys == Keys.None)
            aptfDest[1] = pt;
        else if (ModifierKeys == Keys.Shift)
            aptfDest[3] = pt;
        else
            return;
    }
    else
    return;

    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{  
    Graphics grfx = pea.Graphics;
    GraphicsPath pathWarped = (GraphicsPath) path.Clone();
    WarpMode wm = (WarpMode) miWarpMode.Index;
pathWarped.Warp(aptfDest, path.GetBounds(), new Matrix(), wm);
grfx.DrawPath(new Pen(ForeColor), pathWarped);

} }

Use the left and right mouse buttons to set the upper left and upper right destination coordinates. Use the left and right mouse buttons with the Shift key pressed to set the lower left and lower right destination coordinates. Use the menu to select between Perspective and Bilinear modes. (And notice the clever way in which the OnPaint method casts the Index property of the clicked menu item to a member of type WarpMode.) Here's an example of a Perspective warp:

The path provides a convenient way for you to implement your own nonlinear transforms. You first store the figure you want to display in a path. You then access the PathPoints and PathTypes properties of the path to obtain all the coordinate points. Modify these points in whatever way you want, and then use one of the nondefault GraphicsPath constructors to create a new path based on the modified arrays. I have two examples of this technique in Chapter 19.

**Clipping with Paths**

Besides drawing and filling paths, you can also use paths to set a clipping region for the Graphics object:

**Graphics SetClip Methods (selection)**

```csharp
void SetClip(GraphicsPath path)
void SetClip(GraphicsPath path, CombineMode cm)
```

Suppose the path contains an ellipse. When you call the first version of SetClip, all subsequent drawing is restricted to that ellipse. I'll talk about the second version of SetClip shortly. But first, let's jump right into a demonstration program. The Clover program defines a path containing four overlapping ellipses and uses that for the clipping region.

**Clover.cs**

```csharp
// Clover.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Clover : PrintableForm
{
    public new static void Main()
    {
```
Application.Run(new Clover());

public Clover()
{
    Text = "Clover";
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();

    path.AddEllipse(0, cy / 3, cx / 2, cy / 3); // Left
    path.AddEllipse(cx / 2, cy / 3, cx / 2, cy / 3); // Right
    path.AddEllipse(cx / 3, 0, cx / 3, cy / 2); // Top
    path.AddEllipse(cx / 3, cy / 2, cx / 3, cy / 2); // Bottom

    grfx.SetClip(path);
    grfx.TranslateTransform(cx / 2, cy / 2);

    Pen pen = new Pen(clr);
    float fRadius = (float) Math.Sqrt(Math.Pow(cx / 2, 2) + Math.Pow(cy / 2, 2));

    for (float fAngle = 0; fAngle < (float) Math.PI * 2; fAngle += (float) Math.PI / 180)
    {
        grfx.DrawLine(pen, 0, 0, fRadius * (float) Math.Cos(fAngle), -fRadius * (float) Math.Sin(fAngle));
    }
}

The GraphicsPath is created in the DoPage method. The path consists of four ellipses based on the size of the client area or the printer page. The SetClip method sets the clipping region for the Graphics object based on the path.

The DoPage method next sets an origin in the center of the drawing space and draws 360 lines radiating from the center. These lines are clipped to the interior of the ellipses:
Such an image would be difficult to draw in any other way. You'll notice that the clipping region
doesn't include the area where the ellipses overlap. That's a result of using the default path-filling
mode of FillMode.Alternate. If you change the filling mode to
path.FillMode = FillMode.Winding;

before calling SetClip, those overlapping areas become part of the clipping region as well.

Clipping is often an algorithmically slow process. I've derived the Clover class from PrintableForm so
that you can click on the client area and print the image, but be forewarned that it could take an hour
or more for the program to print.

The question naturally arises, How do the page transform and the world transform affect the clipping
region?

When you call SetClip, the path coordinates are assumed to be world coordinates. The world
coordinates are converted to device coordinates just as if you were drawing or filling the path. The
clipping region is saved in device coordinates and remains in device coordinates. For example, after
the SetClip call in Clover, you can change the page transform and the world transform to anything,
and drawing will still be restricted to the same area of the window. In fact, I've used
TranslateTransform in Clover without affecting the location of the clipping region.

The second version of SetClip I showed lets you combine the existing clipping region with the new
clipping region specified in the SetClip method:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace</td>
<td>0</td>
<td>Clip = New</td>
</tr>
<tr>
<td>Intersect</td>
<td>1</td>
<td>Clip = New ∧ Existing</td>
</tr>
<tr>
<td>Union</td>
<td>2</td>
<td>Clip = New ∨ Existing</td>
</tr>
<tr>
<td>Xor</td>
<td>3</td>
<td>Clip = Union – Intersection</td>
</tr>
<tr>
<td>Exclude</td>
<td>4</td>
<td>Clip = Existing – New</td>
</tr>
<tr>
<td>Complement</td>
<td>5</td>
<td>Clip = New – Existing</td>
</tr>
</tbody>
</table>

The following program creates a clipping region based on two overlapping ellipses. A menu item lets
you select which CombineMode value is used to combine the two ellipses. The program then colors
its entire client area. As in the PathWarping program, I've used the submenu indices (which range
from 0 through 5) as the CombineMode value.
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClippingCombinations : PrintableForm
{
    string strCaption = "CombineMode = ";
    MenuItem miCombineMode = null;

    public new static void Main()
    {
        Application.Run(new ClippingCombinations());
    }

    public ClippingCombinations()
    {
        Text = strCaption + (CombineMode)0;

        Menu = new MainMenu();
        Menu.MenuItems.Add("&CombineMode");

        EventHandler ehClick = new EventHandler(MenuCombineModeOnClick);
        for (int i = 0; i < 6; i++)
        {
            MenuItem mi = new MenuItem("&" + (CombineMode)i);
            mi.Click += ehClick;
            mi.RadioCheck = true;

            Menu.MenuItems[0].MenuItems.Add(mi);
        }
        miCombineMode = Menu.MenuItems[0].MenuItems[0];
        miCombineMode.Checked = true;
    }

    void MenuCombineModeOnClick(object obj, EventArgs ea)
    {
        miCombineMode.Checked = false;
        miCombineMode = (MenuItem)obj;
        miCombineMode.Checked = true;

        Text = strCaption + (CombineMode)miCombineMode.Index;
    }
}
Invalidate();

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();
    path.AddEllipse(0, 0, 2 * cx / 3, cy);
    grfx.SetClip(path);

    path.Reset();
    path.AddEllipse(cx / 3, 0, 2 * cx / 3, cy);
    grfx.SetClip(path, (CombineMode)miCombineMode.Index);

    grfx.FillRectangle(Brushes.Red, 0, 0, cx, cy);
}

Here's the result when the two ellipses are combined with CombineMode.Xor:

![Image of two ellipses combined with CombineMode.Xor]

Additional versions of the SetClip method let you set the clipping region (or combine the clipping region) with a rectangle:

**Graphics SetClip Methods (selection)**

```csharp
default void SetClip(Rectangle rect)
void SetClip(Rectangle rect, CombineMode cm)
void SetClip(RectangleF rectf)
void SetClip(RectangleF rectf, CombineMode cm)
```

The Graphics class also includes methods named IntersectClip and ExcludeClip to modify the existing clipping region. To return the clipping region to normal (that is, an infinitely large region) call the following method:
Graphics ResetClip Method

void ResetClip()

Clipping Bitmaps

Clipping lets you draw nonrectangular areas of a bitmap. Here's a program that loads an image and defines a path in its constructor. In the DoPage method, the program sets a clipping region based on the path and draws the bitmap.

KeyholeClip.cs

//------------------------------------------
// KeyholeClip.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class KeyholeClip: PrintableForm
{
    protected Image image;
    protected GraphicsPath path;

    public new static void Main()
    {
        Application.Run(new KeyholeClip());
    }
    public KeyholeClip()
    {
        Text = "Keyhole Clip";

        image = Image.FromFile(
            "..\..\..\..\Images and Bitmaps\Apollo11FullColor.jpg");

        path = new GraphicsPath();
        path.AddArc(80, 0, 80, 80, 45, -270);
        path.AddLine(70, 180, 170, 180);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.SetClip(path);
        grfx.DrawImage(image, 0, 0, image.Width, image.Height);
    }
}

The result looks a bit incongruous (a keyhole on the moon?), but it works:
Obviously, I defined the path in this program based on this specific image, and under the assumption that the image would be drawn using its pixel dimension with the upper left corner at the point (0, 0).

But suppose you wanted to draw the clipped image in the center of the client area. It's easy to draw the image in the center, but how do you get the path in the center also? One solution is to re-create the path based on the size of the client area. Another solution is to translate the path or to use the following methods that translate the clipping region:

**Graphics TranslateClip Methods**

```csharp
void TranslateClip(int cx, int cy)
void TranslateClip(float cx, float cy)
```

The KeyholeClipCentered program overrides the KeyholeClip program and centers both the clipping region and the path in the client area.

**KeyholeClipCentered.cs**

```csharp
//--------------------------------------------------
// KeyholeClipCentered.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class KeyholeClipCentered: KeyholeClip
{
    public new static void Main()
    {
        Application.Run(new KeyholeClipCentered());
    }
    public KeyholeClipCentered()
    {
        Text += " Centered";
    }
}
```
It's also possible to create a new bitmap based on the size of the clipped image and to use transparency to get the same effect. The KeyholeBitmap program demonstrates this technique.

**KeyholeBitmap.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Windows.Forms;

class KeyholeBitmap : PrintableForm
{
    Bitmap bitmap;

    public new static void Main()
    {
        Application.Run(new KeyholeBitmap());
    }

    public KeyholeBitmap()
    {
        Text = "Keyhole Bitmap";

        // Load image.

        Image image = Image.FromFile(
            "..\..\..\..\..\Images and Bitmaps\Apollo11FullColor.jpg");
        // Create clipping path.
    }

```
GraphicsPath path = new GraphicsPath();
path.AddArc(80, 0, 80, 80, 45, -270);
path.AddLine(70, 180, 170, 180);

// Get size of clipping path.
RectangleF rectf = path.GetBounds();

// Create new bitmap initialized to transparent.
bitmap = new Bitmap((int) rectf.Width, (int) rectf.Height,
                     PixelFormat.Format32bppArgb);

// Create Graphics object based on new bitmap.
Graphics grfx = Graphics.FromImage(bitmap);

// Draw original image on bitmap with clipping.
grfx.SetClip(path);
grfx.TranslateClip(-rectf.X, -rectf.Y);
grfx.DrawImage(image, (int) -rectf.X, (int) -rectf.Y,
               image.Width, image.Height);
grfx.Dispose();
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
grfx.DrawImage(bitmap, (cx - bitmap.Width) / 2,
               (cy - bitmap.Height) / 2,
               bitmap.Width, bitmap.Height);
}

The loading of the image and the creation of the path in the constructor are the same as in the
KeyholeClip program. This program then obtains the size of the path and uses that size to create a
new Bitmap object. The pixel format of the bitmap is specified as Format32bppArgb (which is the
default anyway), and the bitmap is initialized to all zeros, which means that the entire bitmap image
is initially transparent. Anything drawn on the bitmap won't be transparent.

The constructor then obtains a Graphics object for the bitmap and uses the path to set a clipping
region. The problem, however, is that the new bitmap is smaller than the loaded bitmap, so the path
isn't oriented correctly. The TranslateClip method moves the clipping region into place, and
DrawImage (with the same offset factors as TranslateClip) renders the image on the new bitmap.

The DoPage method simply centers the bitmap in the display area. The program could save the new
bitmap as a file as well.

Regions and Clipping

Historically, regions predate the path support in Windows by many years. Regions were available in
Windows 1.0 (which was released in 1985), while paths didn't become available in Windows until
they were introduced in the 32-bit versions, beginning with Windows NT 3.1 in 1993 and Windows 95 in 1995.

With the introduction of paths, regions have become much less important in Windows graphics programming. They might even be ignored altogether if not for the role they play in clipping. Basically, when you define a path for clipping, the path is converted into a region. So the deeper you get into clipping, the more you'll have to learn about regions.

As you know, a graphics path is a collection of lines and curves. A region describes an area of the output device. It's fairly straightforward to convert a path to a region. In fact, one of the constructors of the Region class (which is defined in System.Drawing) creates a region directly from a path:

Region Constructors (selection)
Region(GraphicsPath path)

For the purpose of this constructor, all open subpaths are closed. The region encompasses the interiors of all the subpaths in the path. If the subpaths have overlapping areas, the filling mode of the path determines which interior areas become part of the region and which ones do not. Only one method of the Graphics class uses a region for drawing:

Graphics FillRegion Method
void FillRegion(Brush brush, Region rgn)

If the region was created from a path, this method is equivalent to calling FillPath on the original path.

Only one version of the SetClip method uses a region directly:

Graphics SetClip Methods (selection)
void SetClip(Region rgn, CombineMode cm)

It may seem odd that there's no version of SetClip that has a region argument without any CombineMode argument. That's because the Clip property of the Graphics object is itself defined as a Region. Here are three clipping-related properties of Graphics.

Graphics Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region</td>
<td>Clip</td>
<td>get/set</td>
</tr>
<tr>
<td>RectangleF</td>
<td>ClipBounds</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IsClipEmpty</td>
<td>get</td>
</tr>
</tbody>
</table>

So instead of using a method call to set the clipping region from a Region object,
(grfx.SetClip(rgn); // Doesn't exist!

you just set the property:
grfx.Clip = rgn;

The ClipBounds property indicates the smallest rectangle that encompasses the clipping region; IsClipEmpty indicates whether the clipping region defines a nonexistent area.
Two additional properties of the Graphics path relate to clipping:

**Graphics Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>RectangleF</td>
<td>VisibleClipBounds</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IsVisibleClipEmpty</td>
<td>get</td>
</tr>
</tbody>
</table>

With a new Graphics object, the VisibleClipBounds property indicates the size of the drawing surface. For a form, that's the size of the client area; for the printer, it's the size of the printable area of the page. The ClipBounds property indicates an “infinite” boundary rectangle. (Actually, it's not really infinite. It's just extremely large.)

When you set a clipping region for the Graphics object, VisibleClipBounds will be equal to the intersection of the original VisibleClipBounds and the ClipBounds property. If the clipping region is entirely within the display area, VisibleClipBounds and ClipBounds will be equal.

If IsClipEmpty is true, IsVisibleClipEmpty will also be true. However, it could be that IsClipEmpty is false but the clipping region is outside the boundaries of the client area (or printable area of the printer page). In that case, IsVisibleClipEmpty will be true because no part of the clipping region is within the display area.
Chapter 16: Dialog Boxes

Overview

Given that you can decorate an application's main form with buttons and other controls in the same way that you can design a dialog box, what makes a form different from a dialog box? In terms of managing events from the dialog box and its child controls, the difference is slight. Dialog boxes once represented a big conceptual leap in Windows programming. In the Windows Forms library, however, there's not even a separate class for dialog boxes. You simply create another instance (or in most cases, another subclass) of `Form`.

Dialog boxes are either modal or modeless. Modal dialog boxes are the most common. As the name suggests, a modal dialog box changes the mode of input from the main application form to the dialog box. When your program displays a modal dialog box, the user can't switch between the dialog box and another form in your program. The user must explicitly end the dialog box, usually by clicking a push button marked OK (or Open or Save) or a button marked Cancel. The user can, however, switch to another program while the dialog box is still displayed. Some dialog boxes (called system modal) don't even allow switching to other programs: system modal dialog boxes report serious problems and must be ended before the user can do anything else in Windows. (It's not possible to create a system modal dialog box using the Windows Forms library.)

Modeless dialog boxes are much like additional forms the program creates. (The TwoForms and PaintTwoForms programs in Chapter 2 create two forms.) You can switch among the modeless dialog boxes a program creates and the program's main application form.

Programmers often use modal dialog boxes when a program needs to obtain information from a user beyond what can be easily managed in a menu. (Remember the FontMenu program in Chapter 14?) The dialog box often defines fields or properties that allow the program to initialize the dialog box and (ultimately) obtain information from it.

Very often, programmers working with object-oriented languages find it difficult to decide what should be an object. One approach is to design your dialog boxes so that they have a single field (or property) that an application uses to get all the information from the dialog box. That means that the dialog box is associated with a specific object. There are worse ways to structure your programs!

Toward the end of this chapter, I discuss what are known as the common dialog boxes. These are the predefined dialog boxes that you can present to a user for the selection of files, fonts, or colors. As you'll see, each of these dialog boxes often returns a single object.

Your First Modal Dialog Box

Dialog boxes differ from application forms mostly in the way in which they are invoked and (just as important) terminated. This SimpleDialog program demonstrates both of these jobs.

```
class SimpleDialog: Form
{
    string strDisplay = "";

    public static void Main()
    {
```
Application.Run(new SimpleDialog());

public SimpleDialog()
{
    Text = "Simple Dialog;"

    Menu = new MainMenu();
    Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
}

void MenuOnClick(object obj, EventArgs ea)
{
    SimpleDialogBox dlg = new SimpleDialogBox();

    dlg.ShowDialog();

    strDisplay = "Dialog box terminated with " +
        dlg.DialogResult + "!";
    Invalidate();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
}

class SimpleDialogBox: Form
{
    public SimpleDialogBox()
    {
        Text = "Simple Dialog Box;"

        // Standard stuff for dialog boxes

        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox = false;
        MaximizeBox = false;
        MinimizeBox = false;
        ShowInTaskbar = false;

        // Create OK button.

        Button btn = new Button();
        btn.Parent = this;
        btn.Text = "OK";
        btn.Location = new Point(50, 50);
btn.Size = new Size(10 * Font.Height, 2 * Font.Height);
btn.Click += new EventHandler(ButtonOkOnClick);

// Create Cancel button.
btn = new Button();
btn.Parent = this;
btn.Text = "Cancel";
btn.Location = new Point(50, 100);
btn.Size = new Size(10 * Font.Height, 2 * Font.Height);
btn.Click += new EventHandler(ButtonCancelOnClick);
}

void ButtonOkOnClick(object obj, EventArgs ea)
{
    DialogResult = DialogResult.OK;
}

void ButtonCancelOnClick(object obj, EventArgs ea)
{
    DialogResult = DialogResult.Cancel;
}

The program contains two classes; both are derived from Form. The first class, named SimpleDialog, is the class for the program’s main window. The SimpleDialogBox class is the class for the program’s dialog box.

The Main method creates only an instance of the SimpleDialog class. The constructor of this class creates a very small menu containing just one item labeled “Dialog!” When you click this menu item, the MenuOnClick method invokes the dialog box. It begins this job by creating an instance of SimpleDialogBox:

SimpleDialogBox dlg = new SimpleDialogBox();

Although my dialog boxes will tend to be based on classes with long names such as SimpleDialogBox, I'll generally use a variable named dlg or something similar to refer to the dialog box.

When the program creates an object of type SimpleDialogBox, the default constructor defined in the SimpleDialogBox class is executed. That constructor begins by setting the text that will appear in the dialog box caption:

Text = "Simple Dialog Box";

It then sets five additional properties:

FormBorderStyle = FormBorderStyle.FixedDialog;
ControlBox = false;
MaximizeBox = false;
MinimizeBox = false;
ShowInTaskbar = false;

Setting these five properties is common with dialog boxes. The FixedDialog border style doesn’t allow resizing the dialog box, and the next three properties eliminate the control box (also known as the system menu), the maximize box, and the minimize box from the caption bar. Setting ControlBox to false also eliminates the close box. The caption bar contains only the dialog box text (in this case, "Simple Dialog Box"). Although some dialog boxes in Windows have no caption bar, it’s best to use a
caption bar so that the user has the option to move the dialog box to another location of the screen. Finally, you set the fifth property because you don't want the dialog box showing up in the Windows taskbar. The taskbar should be reserved for applications.

The constructor continues by creating two push buttons with the `Text` properties "OK" and "Cancel." Each button is associated with its own handler for the button's `Click` events.

The dialog box isn't visible yet! After the constructor in `SimpleDialogBox` finishes up, the code in the `MenuOnClick` method of `SimpleDialog` calls the `ShowDialog` method of the dialog box:

```csharp
dlg.ShowDialog();
```

`ShowDialog` causes the dialog box to become visible. The `ShowDialog` method doesn't return until the dialog box is terminated.

`ShowDialog` is the method you must use to invoke a modal dialog box. During the time a modal dialog box is displayed, you can't switch back to the program's main form. That's what it means to be modal. (As I said earlier, you can, however, switch to other applications running under Windows.) While the modal dialog box is displayed, the application form can't receive any keyboard or mouse input. However, the form can continue to receive `Tick` events from a `Timer` object and calls to `OnPaint`.

The two buttons in the dialog box have `Click` event handlers named `ButtonOkOnClick` and `ButtonCancelOnClick`. Both methods have just a single line that sets a property of the dialog box form, named `DialogResult`:

<table>
<thead>
<tr>
<th>Form Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>DialogResult</td>
</tr>
</tbody>
</table>

The `DialogResult` property must be set to one of the following enumeration values:

<table>
<thead>
<tr>
<th>DialogResult Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>OK</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
<tr>
<td>Abort</td>
</tr>
<tr>
<td>Retry</td>
</tr>
<tr>
<td>Ignore</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

You'll notice that these members correspond to text strings commonly displayed on buttons within a dialog box. If this table looks familiar, it's because you first encountered it in Chapter 2. The `Show` method of the `MessageBox` class returns a member of the `DialogResult` enumeration.

In the program at hand, the button labeled "OK" sets `DialogResult` to `DialogResult.OK`, and the button labeled "Cancel" sets the property to `DialogResult.Cancel`.

What happens in either case is quite dramatic: the dialog box is closed. It disappears from the screen. The `ShowDialog` method that originally invoked the dialog box now returns control to the `MenuOnClick` method.

Although the dialog box has been terminated and is no longer visible, the dialog box object named `dlg` in the application form is still valid. That means that the `MenuOnClick` method can access the dialog box's `DialogResult` property to determine how the dialog box was terminated. In this particular
case, the MenuOnClick method simply sets the strDisplay field variable and invalidates the form. The OnPaint method displays this string.

Now let’s look at a couple shortcuts. First, the ShowDialog method is defined like so:


```
Form Methods (selection)

DialogResult ShowDialog()
```

The return value is the same as the DialogResult property of the dialog box when the dialog box was terminated. So, a program that invokes a dialog box can save the DialogResult when ShowDialog returns:

```
DialogResult dr = dlg.ShowDialog();
```

Or the ShowDialog call can go right into an if statement:

```
if (dlg.ShowDialog() == DialogResult.OK)
{
    ;
}
else
{
    ;
}
```

or a switch statement:

```
switch(dlg.ShowDialog())
{
    case DialogResult.OK:
        ;
    case DialogResult.Cancel:
        ;
    default:
        ;
}
```

Generally, a program gets information from a dialog box if DialogResult is OK and just continues on its merry way if DialogResult is Cancel.

**Modal Dialog Box Termination**

You’re probably fairly happy with the code I showed in the SimpleDialogBox program. How much easier could it be to terminate a dialog box than just setting a DialogResult property in a button’s Click event handler?

Well, keep reading.

As you probably know from experience, modal dialog boxes are almost always terminated when the user presses a push button. For that reason, the Button class—or more precisely, the IButtonControl interface that Button and LinkLabel implement—also includes a property named DialogResult:

```
IButtonControl Property

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogResult</td>
<td>DialogResult</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```
We've already seen that `Form` has a property named `DialogResult`, and now you see that `Button` has a property named `DialogResult` as well. Usually when you see the same property implemented in both the `Form` and `Button` classes, you'd naturally assume that both classes inherit the property from `Control`. But that's not the case with `DialogResult`, although the two implementations of the property are related.

When you set the `DialogResult` property of a button, you are essentially instructing the button to set the same `DialogResult` of its parent (the dialog box itself) when the button is clicked. The result is that you don't have to install event handlers for the dialog box buttons, as the SimplerDialog program demonstrates.

**SimplerDialog.cs**
```
//SimplerDialog.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class SimplerDialog: Form
{
    string strDisplay = "";

    public static void Main()
    {
        Application.Run(new SimplerDialog());
    }
    public SimplerDialog()
    {
        Text = "Simpler Dialog";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
    }
    void MenuOnClick(object obj, EventArgs ea)
    {
        SimplerDialogBox dlg = new SimplerDialogBox();
        DialogResult dr = dlg.ShowDialog();

        strDisplay = "Dialog box terminated with " + dr + "!";
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
    }
}
```
class SimplerDialogBox: Form
{
    public SimplerDialogBox()
    {
        Text = "Simpler Dialog Box";

        // Standard stuff for dialog boxes
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox     = false;
        MaximizeBox    = false;
        MinimizeBox    = false;
        ShowInTaskbar  = false;

        // Create OK button.

        Button btn = new Button();
        btn.Parent    = this;
        btn.Text      = "OK";
        btn.Location  = new Point(50, 50);
        btn.Size      = new Size(10 * Font.Height, 2 * Font.Height);
        btn.DialogResult = DialogResult.OK;

        // Create Cancel button.

        btn = new Button();
        btn.Parent    = this;
        btn.Text      = "Cancel";
        btn.Location  = new Point(50, 100);
        btn.Size      = new Size(10 * Font.Height, 2 * Font.Height);
        btn.DialogResult = DialogResult.Cancel;
    }
}

This version of the program behaves the same way as SimpleButton when you press the OK or Cancel button. The dialog box doesn't need to explicitly set the form's DialogResult property unless you want to terminate the dialog box by means other than a button.

If you still need to do a little processing when the user presses the OK or Cancel button, you can always install Click event handlers as well. But for purposes of terminating the dialog box, you certainly don't have to.

Accept and Cancel

The dialog boxes created so far are missing a small piece of the normal keyboard interface for dialog boxes. What's there works fine: using the keyboard Tab or arrow keys, you can move input focus between the OK and Cancel buttons. As you move the input focus, the button with the input focus also becomes the default button. You can trigger the button with the input focus by using the spacebar; you can also trigger the default button by using the Enter key.
What you can't do is terminate the dialog box with the Esc key. The Esc key is supposed to be the equivalent of pressing the Cancel button.

Moreover, if you put another type of control in the dialog box (a CheckBox perhaps), you'll find that whenever the CheckBox has the input focus, there is no default push button. Pressing Enter does nothing in that case. If any non-Button control has the input focus, the OK button (or the equivalent of the OK button, labeled Open or Save or something else) is supposed to be the default push button, which means that it should respond to the Enter key.

You can take care of this aspect of the user interface by using the following two properties of Form:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>IButtonControl</td>
<td>AcceptButton</td>
<td>get/set</td>
</tr>
<tr>
<td>IButtonControl</td>
<td>CancelButton</td>
<td>get/set</td>
</tr>
</tbody>
</table>

You can set these two properties to an object of any class that implements IButtonControl, which is probably a Button or LinkLabel object.

The AcceptButton property indicates which Button control should be triggered whenever a non-Button control has the input focus and the Enter key is pressed. Regardless of how you set AcceptButton, any control that implements the IButtonControl interface will become the default button and respond to Enter if it has the input focus.

The CancelButton property indicates the Button control that should be triggered whenever the Esc key is pressed.

You may be curious about IButtonControl. You've already seen a third of what it means to implement the IButtonControl interface, which is to implement a property named DialogResult. The other two-thirds of IButtonControl are these two methods:

**IButtonControl Methods**

```csharp
void NotifyDefault(bool bDefault)
void PerformClick()
```

When a control implementing IButtonControl gets a call to NotifyDefault with an argument of true, it is responsible for visually indicating that it is the default control (and hence will respond to the Enter key). A button indicates that it's the default control with a bold outline. The PerformClick method simulates a button click. That's the method of the default Button control that the form calls when the Enter key is pressed.

Generally, the DialogResult property of the Button control and the AcceptButton and CancelButton properties of the dialog box form go together. For example, when a form creates a button labeled OK or Load or Save, it sets the DialogResult property like so:

```csharp
btn.DialogResult = DialogResult.OK;
```

It also sets the AcceptButton property of the dialog box form to the Button object:

```csharp
AcceptButton = btn;
```

Similarly, when a form creates a Cancel button, it sets the DialogResult like this:

```csharp
btn.DialogResult = DialogResult.Cancel;
```

And it sets the CancelButton property of the form like so:

```csharp
CancelButton = btn;
```

I'll show you a program that sets these properties shortly.
Screen Location

You’ve probably noticed that newly launched Windows applications often appear in different locations of the screen. When a Windows session first begins, Windows positions the first application in the upper left corner of the screen. Each successive application is then positioned somewhat to the right of and below the previous one in a cascaded pattern using an offset equal to $SystemInformation.CaptionButtonSize$ plus $SystemInformation.FrameBorderSize$.

While this behavior is fine for applications, the same rules also apply to dialog boxes, with less than optimal results. The result is that a dialog box could appear some distance from the application that invokes it. The problem may not be so evident in the SimpleDialog and SimplerDialog programs because you’re probably running the programs, invoking the dialog box, closing the dialog box, and then closing the program. In that chain of events, the dialog box often appears suitably offset from the application form. But if you were to run a couple other programs before you invoke the dialog box, the results would be different.

You can override the default behavior governing the location of forms by using the $StartPosition$ property:

<table>
<thead>
<tr>
<th>Form Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>FormStartPosition</td>
</tr>
</tbody>
</table>

$FormStartPosition$ is an enumeration:

<table>
<thead>
<tr>
<th>$FormStartPosition$ Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Manual</td>
</tr>
<tr>
<td>CenterScreen</td>
</tr>
<tr>
<td>WindowsDefaultLocation</td>
</tr>
<tr>
<td>WindowsDefaultBounds</td>
</tr>
<tr>
<td>CenterParent</td>
</tr>
</tbody>
</table>

The default for Windows Forms applications is $WindowsDefaultLocation$, which means that Windows positions the form as I described but the application itself sizes the form. Actually, the constructor for $Form$ sets the size of the form, but the constructor in any class that inherits from $Form$ can override that size. That’s why all Windows Forms applications have the same default size, and this size is different from regular (non–Windows Forms) Windows programs that use a default size.

You can get the regular Windows default position and size by specifying $WindowsDefaultBounds$, which means that Windows sets both the location and size of the form. When you use $WindowsDefaultBounds$, any attempt to set the location or size of the form in your program’s constructor will be ignored; when you use $WindowsDefaultLocation$, your constructor can set a size but not a location.

The $CenterParent$ option allows a program to position a dialog box in the center of the program’s form without doing any calculations. This placement might not be optimum, however. So far in this chapter, I haven’t been resizing application client areas or dialog boxes, so $CenterParent$ results in a dialog box that exactly overlays (and completely hides) the application that invoked it. $CenterScreen$ positions a dialog box in the center of the screen and is useful for modeless dialog boxes that sometimes appear on the screen while an application is loading. Both $CenterParent$ and $CenterScreen$ allow a dialog box constructor to set its own size.

The $Manual$ option lets a dialog box have complete freedom in setting its position and size. Generally, a dialog box will want to use this option to position itself relative to the form that invoked it. The best way for a dialog box to obtain the form that invoked it is the $ActiveForm$ property.
Here's a program named BetterDialog with a dialog box that sets its location properly offset from the application form. The program also demonstrates the use of the `AcceptButton` and `CancelButton` properties I talked about earlier.

**BetterDialog.cs**

```csharp
//-------------------------------------------
// BetterDialog.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterDialog: Form
{
    string strDisplay = "";

    public static void Main()
    {
        Application.Run(new BetterDialog());
    }

    public BetterDialog()
    {
        Text = "Better Dialog";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Dialog!", new EventHandler(MenuOnClick));
    }

    void MenuOnClick(object obj, EventArgs ea)
    {
        BetterDialogBox dlg = new BetterDialogBox();
        DialogResult dr = dlg.ShowDialog();

        strDisplay = "Dialog box terminated with " + dr + "!";
        Invalidate();
    }

    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        grfx.DrawString(strDisplay, Font, new SolidBrush(ForeColor), 0, 0);
    }
}

class BetterDialogBox: Form
{
    public BetterDialogBox()
    {
        Text = "Better Dialog Box";
    }
```
// Standard stuff for dialog boxes

FormBorderStyle = FormBorderStyle.FixedDialog;
ControlBox = false;
MaximizeBox = false;
MinimizeBox = false;
ShowInTaskbar = false;
StartPosition = FormStartPosition.Manual;
Location = ActiveForm.Location +
    SystemInformation.CaptionButtonSize +
    SystemInformation.FrameBorderSize;

// Create OK button.

Button btn = new Button();
btn.Parent = this;
btn.Text = "OK";
btn.Location = new Point(50, 50);
btn.Size = new Size(10 * Font.Height, 2 * Font.Height);
btn.DialogResult = DialogResult.OK;

AcceptButton = btn;

// Create Cancel button.

btn = new Button();
btn.Parent = this;
btn.Text = "Cancel";
btn.Location = new Point(50, 100);
btn.Size = new Size(10 * Font.Height, 2 * Font.Height);
btn.DialogResult = DialogResult.Cancel;

CancelButton = btn;

Now as part of the standard housekeeping at the beginning of the dialog box constructor, the
StartPosition property is set to FormStartPosition.Manual. The dialog box also sets its Location
property equal to the Location property of the active form (the form that invoked the dialog box) plus
those two SystemInformation properties I mentioned earlier.

Also take note that as the dialog box creates each of the two buttons, it also sets the AcceptButton
and CancelButton properties of the dialog box form. The dialog box now has a full and proper
keyboard interface.

The About Box
One common dialog box found in nearly all nontrivial applications is called an about box. The about box can be as simple as a copyright notice or as complex as a display of system resources and perhaps a phone number and a Web site for tech support.

Here's a program that displays an about box containing an icon, two label controls, and one button.

**AboutBox.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class AboutBox: Form
{
    public static void Main()
    {
        Application.Run(new AboutBox());
    }
    public AboutBox()
    {
        Text = "About Box";
        Icon = new Icon(GetType(), "AboutBox.AforAbout.ico");

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Help");
        Menu.MenuItems[0].MenuItems.Add("&About AboutBox...",
            new
            EventHandler(MenuAboutOnClick));
    }
    void MenuAboutOnClick(object obj, EventArgs ea)
    {
        AboutDialogBox dlg = new AboutDialogBox();
        dlg.ShowDialog();
    }
}

class AboutDialogBox: Form
{
    public AboutDialogBox()
    {
        Text = "About AboutBox";

        StartPosition = FormStartPosition.CenterParent;
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox = false;
        MaximizeBox = false;
        MinimizeBox = false;
    }
```
ShowInTaskbar  = false;

Label label1  = new Label();
label1.Parent = this;
label1.Text   = " AboutBox Version 1.0 ";
label1.Font   = new Font(FontFamily.GenericSerif, 24,
                        FontStyle.Italic);
label1.AutoSize = true;
label1.TextAlign = ContentAlignment.MiddleCenter;

Icon icon = new Icon(GetType(), "AboutBox.AforAbout.ico");

PictureBox picbox = new PictureBox();
picbox.Parent = this;
picbox.Image = icon.ToBitmap();
picbox.SizeMode = PictureBoxSizeMode.AutoSize;
picbox.Location = new Point(label1.Font.Height / 2,
                           label1.Font.Height / 2);

label1.Location = new Point(picbox.Right, label1.Font.Height / 2);

int iClientWidth = label1.Right;

Label label2 = new Label();
label2.Parent = this;
label2.Text   = "\xA9 2001 by Charles Petzold";
label2.Font   = new Font(FontFamily.GenericSerif, 16);
label2.Location = new Point(0, label1.Bottom +
                           label2.Font.Height);
label2.Size    = new Size(iClientWidth, label2.Font.Height);
label2.TextAlign = ContentAlignment.MiddleCenter;

Button button = new Button();
button.Parent = this;
button.Text   = "OK";
button.Size   = new Size(4 * button.Font.Height,
                        2 * button.Font.Height);
button.Location = new Point((iClientWidth - button.Size.Width) / 2,
                          label2.Bottom + 2 *
                          button.Font.Height);

button.DialogResult = DialogResult.OK;

CancelButton = button;
AcceptButton = button;

ClientSize = new Size(iClientWidth,
                  button.Bottom + 2 * button.Font.Height);
}
}

AforAbout.ico

The constructors of both the AboutBox class and the AboutDialogBox class load the icon resource. The AboutBox class sets the resource as the form icon by using the Icon property. The AboutDialogBox class creates a PictureBox control for displaying the icon in the dialog box.

The constructor of the AboutDialogBox class is a bit involved because of the positioning of the controls. For the first Label control (which contains the name of the program in a 24-point italic font), the AutoSize property is set to true. For the PictureBox control, theSizeMode property is set to PictureBoxSizeMode.AutoSize. The picture box is positioned based on the resultant size of the label, and the label is positioned based on the resultant size of the picture box.

For a dialog box with one button, the DialogResult property of the button is usually assigned DialogResult.OK. You should also assign both the AcceptButton and CancelButton properties of the form to that button.

Aside from the button, you don't need to use controls in a dialog box like this. Instead of using Label and PictureBox controls, you can call DrawString and DrawIcon from the OnPaint method in the AboutDialogBox class.

Actually, for a dialog box as simple as this one, you don't even need to derive a class from Form to use for the dialog box. You'll recall in Chapter 2 how forms could be created by making an instance of Form rather than a class derived from Form. You can do the same thing with dialog boxes. In fact, the problem of sharing data between an application form and a dialog box becomes much simpler when the dialog box isn't in a separate form.

Much simpler, yes, but also not quite as structured. Wouldn't it be nice to write dialog boxes that could be reused in other applications? It might not always be possible, but it should always be a goal.

Defining Properties in Dialog Boxes
The RadioButtons program in Chapter 12 shows how you can define a group of radio buttons and a check box to indicate the color of an ellipse and whether the ellipse is outlined or filled. Let's implement the same controls (plus a couple buttons) in a dialog box instead. The dialog box must have some provision that lets a program initialize the controls and also some way for an application to obtain the user's selections. You generally provide this interface through public properties of the dialog box form or, in some cases—particularly if you're in a hurry—through public fields.

This dialog box implements two public properties, named Color and Fill, that give another class access to the two items that the user sets using the dialog box.

ColorFillDialogBox.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorFillDialogBox: Form
{
    protected GroupBox grpbox;
    protected CheckBox chkbox;

    public ColorFillDialogBox()
    {
        Text = "Color/Fill Select";

        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox = false;
        MinimizeBox = false;
        MaximizeBox = false;
        ShowInTaskbar = false;
        Location = ActiveForm.Location +
                    SystemInformation.CaptionButtonSize +
                    SystemInformation.FrameBorderSize;

        string[] astrColor = { "Black", "Blue", "Green", "Cyan",
                              "Red", "Magenta", "Yellow", "White"};

        grpbox = new GroupBox();
        grpbox.Parent = this;
        grpbox.Text = "Color";
        grpbox.Location = new Point(8, 8);
        grpbox.Size = new Size(96, 12 * (astrColor.Length + 1));

        for (int i = 0; i < astrColor.Length; i++)
        {
            RadioButton radiobtn = new RadioButton();
            radiobtn.Parent = grpbox;
            radiobtn.Text = astrColor[i];
            radiobtn.Click += new EventHandler(RadioButton_Click);
            radiobtn.CheckedChanged += new EventHandler(RadioButton_CheckedChanged);
            radiobtn.Check = (i == 0);
            grpbox.Controls.Add(radiobtn);
        }

        chkbox = new CheckBox();
        chkbox.Text = "Fill";
        chkbox.Location = new Point(8, 96);
        chkbox.CheckedChanged += new EventHandler(CheckBox_CheckedChanged);
        chkbox.CheckPosition =ence.
    }

    void CheckBox_CheckedChanged(object sender, EventArgs e)
    {
        ColorFillDialogBox box = (ColorFillDialogBox)sender;
        box.Fill = chkbox.CheckState == CheckState.Checked;
    }

    void RadioButton_CheckedChanged(object sender, EventArgs e)
    {
        ColorFillDialogBox box = (ColorFillDialogBox)sender;
        box.Color = ((RadioButton)sender).CheckedIndex;
    }

    void RadioButton_Click(object sender, EventArgs e)
    {
        ColorFillDialogBox box = (ColorFillDialogBox)sender;
        box.Color = ((RadioButton)sender).CheckedIndex;
    }
}
```

radiobtn.Text = astrColor[i];
radiobtn.Location = new Point(8, 12 * (i + 1));
radiobtn.Size = new Size(80, 10);
}
chkbox = new CheckBox();
chkbox.Parent = this;
chkbox.Text = "Fill Ellipse";
chkbox.Location = new Point(8, grpbox.Bottom + 4);
chkbox.Size = new Size(80, 10);

Button btn = new Button();
btn.Parent = this;
btn.Text = "OK";
btn.Location = new Point(8, chkbox.Bottom + 4);
btn.Size = new Size(40, 16);
btn.DialogResult = DialogResult.OK;
AcceptButton = btn;

btn = new Button();
btn.Parent = this;
btn.Text = "Cancel";
btn.Location = new Point(64, chkbox.Bottom + 4);
btn.Size = new Size(40, 16);
btn.DialogResult = DialogResult.Cancel;
CancelButton = btn;

ClientSize = new Size(112, btn.Bottom + 8);
AutoSizeBaseSize = new Size(4, 8);
}
public Color Color
{
    get
    {
        for (int i = 0; i < grpbox.Controls.Count; i++)
        {
            RadioButton radiobtn = (RadioButton)grpbox.Controls[i];

            if (radiobtn.Checked)
                return Color.FromName(radiobtn.Text);
        }
        return Color.Black;
    }
    set
    {
    }
for (int i = 0; i < grpbox.Controls.Count; i++)
{
    RadioButton radiobtn = (RadioButton) grpbox.Controls[i];

    if (value == Color.FromName(radiobtn.Text))
    {
        radiobtn.Checked = true;
        break;
    }
}

public bool Fill
{
    get { return checkbox.Checked; }  
    set { checkbox.Checked = value; }
}

The constructor uses a classical dialog box coordinate system to create, position, and size all the controls. The code is flexible enough to let you put additional colors in the astrColor array.

The class stores the GroupBox and CheckBox objects as protected fields, but it doesn't keep track of the current state of the radio buttons or check boxes. The class relies on the auto-check facility of these two controls to keep the check state consistent with the user's selections. But it's still necessary for the class to provide an interface to the state of the controls.

That interface is provided in the public Color and Fill properties defined toward the bottom of the program. The Fill property is quite simple. The get accessor returns the Checked property of the check box; the set accessor sets that property. The Color property, however, requires a search through all the radio buttons, which is equivalent to a search through all the children of the group box control, which is also equivalent to looping through the Controls property of the group box. The get accessor returns a Color object represented by the currently checked radio button; the set accessor checks the radio button corresponding to a particular Color object.

Here's what the dialog box looks like (although we haven't yet encountered a program that invokes it):
Now this is not a dialog box that a whole lot of different applications require, but it's written to allow reuse. That's the goal.

The DrawOrFillEllipse program implements a menu item to invoke this dialog box and then uses the results to update its client area with a colored (and possibly filled) ellipse.

**DrawOrFillEllipse.cs**

```csharp
//------------------------------------------------
// DrawOrFillEllipse.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DrawOrFillEllipse: Form
{
    Color colorEllipse = Color.Red;
    bool bFillEllipse = false;

    public static void Main()
    {
        Application.Run(new DrawOrFillEllipse());
    }
    public DrawOrFillEllipse()
    {
        Text = "Draw or Fill Ellipse";
        ResizeRedraw = true;

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Options");
    }
}
```
I want to draw your attention to the `MenuColorOnClick` method, which is the event handler associated with the menu item to invoke the dialog box. The method creates an object of type `ColorFillDialogBox`. Keep in mind that you can set just about any property of the dialog box form at this time. You may want to change the `Text` property, for example. A dialog box designed for reuse could specifically implement other properties that let an application program alter its appearance and functionality.

The `MenuColorOnClick` method continues with dialog box initialization by setting the two custom properties implemented in `ColorFillDialogBox` from fields in the `DrawOrFillEllipse` class:

```csharp
dlg.Color = colorEllipse;
dlg.Fill = bFillEllipse;
```

The next statement calls the `ShowDialog` method of the dialog box form, which won't return until the dialog box is closed. At that time, the program compares the return value of `ShowDialog` with `DialogResult.OK`. If the dialog box was terminated with the OK button, the program stores the new values of the properties and invalidates the client area:

```csharp
if (dlg.ShowDialog() == DialogResult.OK)
{
    colorEllipse = dlg.Color;
    bFillEllipse = dlg.Fill;
    Invalidate();
}
```

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Rectangle rect = new Rectangle(0, 0, ClientSize.Width - 1,
        ClientSize.Height - 1);
    if (bFillEllipse)
        grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
    else
        grfx.DrawEllipse(new Pen(colorEllipse), rect);
}

Menu.MenuItems[0].MenuItems.Add("&Color...",
    new EventHandler(MenuColorOnClick));

void MenuColorOnClick(object obj, EventArgs ea)
{
    ColorFillDialogBox dlg = new ColorFillDialogBox();

    dlg.Color = colorEllipse;
    dlg.Fill = bFillEllipse;

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        colorEllipse = dlg.Color;
        bFillEllipse = dlg.Fill;
        Invalidate();
    }
}
The MenuColorOnClick method represents very standard code for creating, initializing, invoking, and obtaining information from dialog boxes.

**Implementing an Apply Button**

In recent years, some dialog boxes have sprouted buttons labeled Apply. The Apply button doesn't make the dialog box go away, but it causes the application to use the new settings specified in the dialog box.

The Apply button upsets the normal orderly relationship between an application and a modal dialog box. It requires that the application be informed when the Apply button has been pressed before the ShowDialog method returns control to the application.

It is very tempting to implement an Apply button by defining a public method in your application form that the dialog box form calls when the Apply button is pressed. In fact, you may wonder why this isn't the right way to go. It isn't a good idea because the class implementing the dialog box would then require an application using that class to implement a particular method with a particular name. Do you know of any .NET class that forces an application to define a particular method to use the class?

No you don't. But you're very familiar with the facility by which the .NET classes communicate to applications, and that is by events.

Let's rewrite the ColorFillDialogBox class so that it includes, first, an Apply button; second, a property to enable and disable that button; and third, an event. This job is going to be easier than you probably fear!

Here's a ColorFillDialogBoxWithApply class that inherits from ColorFillDialogBox. Besides implementing an Apply button, this class must also move the controls around a bit and set a new client size to accommodate the new button.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class ColorFillDialogBoxWithApply: ColorFillDialogBox
{
    Button btnApply;

    public event EventHandler Apply;

    public ColorFillDialogBoxWithApply()
    {
        grpbox.Location = new Point(36, 8);
        chkbox.Location = new Point(36, grpbox.Bottom + 4);

        btnApply = new Button();
        btnApply.Parent = this;
        btnApply.Enabled = false;
        btnApply.Text = "Apply";
    }
}
```
Toward the top of this class, you'll see the statement

```
public event EventHandler Apply;
```

This statement defines a public event named `Apply` that is based on the `EventHandler` delegate.

The class also has a new private field named `btnApply`. Obviously, this is the `Button` object labeled `Apply`, which is created in the new constructor. Notice that the constructor sets the `Enabled` property for this button to `false`. A program using this dialog box may not want to deal with an Apply button. But then how is the button enabled? By a public property, of course! This class implements a new property named `ShowApply` that lets a program using the dialog box enable or disable the Apply button at will.

The Apply button isn't associated with any `DialogResult` because the Apply button doesn't terminate the dialog box. Instead, I install an event handler named `ButtonApplyOnClick` for the button's `Click` event. This method contains the magic code required to implement an event in a class. Here's what's executed whenever the user triggers the Apply button:

```
if (Apply != null)
    Apply(this, new EventArgs());
```

The `if` statement checks whether at least one handler has been installed for the `Apply` event. If so, all the installed event handlers are called with the `EventHandler` arguments: the first argument indicates the origin of the event (the dialog box form), and the second argument is an object of type `EventArgs`.

To be even more consistent with the .NET classes, the `ButtonApplyOnClick` method would call a protected virtual method named `OnApply` with a single `EventArgs` argument. `OnApply` would then contain the code to call the `Apply` event handlers.

And here's the program that uses the new version of the dialog box.

```
DrawOrFillEllipseWithApply.cs
//----------------------------------------------------------------------
// DrawOrFillEllipseWithApply.cs © 2001 by Charles Petzold
//----------------------------------------------------------------------
using System;
```
using System.Drawing;
using System.Windows.Forms;

class DrawOrFillEllipseWithApply: Form
{
    Color colorEllipse = Color.Red;
    bool bFillEllipse = false;

    public static void Main()
    {
        Application.Run(new DrawOrFillEllipseWithApply());
    }
    public DrawOrFillEllipseWithApply()
    {
        Text = "Draw or Fill Ellipse with Apply";
        ResizeRedraw = true;

        Menu = new MainMenu();
        Menu.MenuItem.Add("&Options");
        Menu.MenuItem[0].MenuItem.Add("&Color...",
                new EventHandler(MenuColorOnClick));
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        ColorFillDialogBoxWithApply dlg =
                new ColorFillDialogBoxWithApply();
        dlg.ShowApply = true;
        dlg.Apply += new EventHandler(ColorFillDialogOnApply);
        dlg.Color = colorEllipse;
        dlg.Fill = bFillEllipse;

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            colorEllipse = dlg.Color;
            bFillEllipse = dlg.Fill;
            Invalidate();
        }
    }
    void ColorFillDialogOnApply(object obj, EventArgs ea)
    {
        ColorFillDialogBoxWithApply dlg =
                (ColorFillDialogBoxWithApply) obj;
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Rectangle rect = new Rectangle(0, 0, ClientSize.Width - 1,
                                    ClientSize.Height - 1);
    if(bFillEllipse)
        grfx.FillEllipse(new SolidBrush(colorEllipse), rect);
    else
        grfx.DrawEllipse(new Pen(colorEllipse), rect);
}

During the MenuColorOnClick method, the program enables the Apply button and installs an event handler for the button:

dlg.ShowApply = true;
dlg.Apply += new EventHandler(ColorFillDialogOnApply);

The ColorFillDialogOnApply event handler casts the object argument to an object of type ColorFillDialogBoxWithApply in order to get access to the Color and Fill properties. The program then sets its fields from the properties and invalidates the client area. As I mentioned earlier, even if a modal dialog box is active, an application form can still get Paint events. So, the client area is able to update itself based on the new dialog box settings.

The Modeless Dialog Box

At the beginning of this chapter, I mentioned that dialog boxes can be either modal or modeless. So far we've been looking at modal dialog boxes, certainly the more common of the two types. Modeless dialog boxes allow the user to switch between the dialog box and the form that created it.

Modeless dialog boxes are preferred when the user would find it convenient to keep the dialog box displayed for a while. Perhaps the most common modeless dialog boxes are the Find and Replace dialog boxes displayed by word processing programs. As a user, you probably want to keep such a dialog box active for a while so that you can do multiple find or replace actions. Yet while the dialog box is active, you also want to be able to edit the document on which you're running the find or replace.

The modeless dialog box implemented in this next class is based on the ColorScroll program in Chapter 12. The form contains three scroll bars and six labels that resize themselves based on the form's size. The dialog box is intended to remain active during the entire duration of the program that displays it; thus, it has no buttons or a close box on its caption bar. A property named Color provides public access to the scroll bar values.
{  
    Label[] alabelName = new Label[3];
    Label[] alabelValue = new Label[3];
    VScrollBar[] avscroll = new VScrollBar[3];

    public event EventHandler Changed;

    public ColorScrollDialogBox()
    {
        Text = "Color Scroll Dialog Box";

        ControlBox = false;
        MinimizeBox = false;
        MaximizeBox = false;
        ShowInTaskbar = false;

        Color[] acolor = { Color.Red, Color.Green, Color.Blue };  

        for (int i = 0; i < 3; i++)
        {
            alabelName[i] = new Label();
            alabelName[i].Parent = this;
            alabelName[i].ForeColor = acolor[i];
            alabelName[i].Text = "&" + acolor[i].ToKnownColor();
            alabelName[i].TextAlign = ContentAlignment.MiddleCenter;

            avscroll[i] = new VScrollBar();
            avscroll[i].Parent = this;
            avscroll[i].SmallChange = 1;
            avscroll[i].LargeChange = 16;
            avscroll[i].Minimum = 0;
            avscroll[i].Maximum = 255 + avscroll[i].LargeChange - 1;
            avscroll[i].ValueChanged +=
            new EventHandler(ScrollOnValueChanged);
            avscroll[i].TabStop = true;

            alabelValue[i] = new Label();
            alabelValue[i].Parent = this;
            alabelValue[i].TextAlign = ContentAlignment.MiddleCenter;
        }

        OnResize(EventArgs.Empty);
    }

    public Color Color
    {
    
    }
get
{
    return Color.FromArgb(avscroll[0].Value,
                          avscroll[1].Value,
                          avscroll[2].Value);
}
set
{
    avscroll[0].Value = value.R;
    avscroll[1].Value = value.G;
    avscroll[2].Value = value.B;
}
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);
    int cx = ClientSize.Width;
    int cy = ClientSize.Height;
    int cyFont = Font.Height;
    for (int i = 0; i < 3; i++)
    {
        alabelName[i].Location = new Point(i * cx / 3, cyFont / 2);
        alabelName[i].Size = new Size(cx / 3, cyFont);
        avscroll[i].Location = new Point((4 * i + 1) * cx / 12,
                                         2 * cyFont);
        avscroll[i].Size = new Size(cx / 6, cy - 4 * cyFont);
        alabelValue[i].Location = new Point(i * cx / 3,
                                             cy - 3 * cyFont / 2);
        alabelValue[i].Size = new Size(cx / 3, cyFont);
    }
}
void ScrollOnValueChanged(Object obj, EventArgs ea)
{
    for (int i = 0; i < 3; i++)
    {
        if ((VScrollBar) obj == avscroll[i])
            alabelValue[i].Text = avscroll[i].Value.ToString();
        if (Changed != null)
            Changed(this, new EventArgs());
    }
}
As we've seen, modal dialog boxes don't usually need to implement their own public events unless they include an Apply button. However, modeless dialog boxes almost always need to actively communicate with the application that invokes them, and the best way to do this is through events.

The ColorScrollDialogBox class implements an event named Changed that is triggered whenever one of the scroll bars has a ValueChanged event. If I wanted to imitate the .NET Framework more, I'd also include a protected virtual OnChanged method in the class. The OnChanged method would be called by ScrollOnValueChanged, and OnChanged would then call the Changed event.

You don't use ShowDialog with a modeless dialog box. ShowDialog doesn't return until the dialog box is closed, and that's not what you want. Instead, you use Show, a method of Form we encountered in Chapter 2 and haven't seen since.

Another crucial part of implementing modeless dialog boxes involves this property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Owner</td>
<td>get/set</td>
</tr>
</tbody>
</table>

You set the Owner property of the modeless dialog box to the application form. Doing so causes the form to own the dialog box. Being owned by an application form means that the modeless dialog box will always appear in front of the application form. Also, whenever the application form is minimized, the modeless dialog box disappears from the screen.

The following program creates an object of type ColorScrollDialogBox, sets the Owner property to itself (the application form), initializes the Color property of the dialog box with the application form's BackColor property, sets an event handler for the dialog box's Changed event, and then calls the dialog box's Show method. The dialog box stays active (and lets you change the application form's background color) until the application is terminated.

ModelessColorScroll.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ModelessColorScroll: Form
{
    public static void Main()
    {
        Application.Run(new ModelessColorScroll());
    }
    public ModelessColorScroll()
    {
        Text = "Modeless Color Scroll";

        ColorScrollDialogBox dlg = new ColorScrollDialogBox();

        dlg.Owner = this;
        dlg.Color = BackColor;
        dlg.Changed += new EventHandler(ColorScrollOnChanged);
        dlg.Show();
    }
}
```
void ColorScrollOnChanged(object obj, EventArgs ea)
{
    ColorScrollDialogBox dlg = (ColorScrollDialogBox) obj;

    BackColor = dlg.Color;
}

The Transform program in Chapter 18 demonstrates another modeless dialog box (which itself invokes a modal dialog box) that lets you interactively set the six elements of a matrix transform.

The Common Dialog Boxes

One of the primary goals of Windows has always been to promote a standardized user interface. In the early days of Windows, some user interface conventions were established fairly quickly. Almost every software manufacturer adopted the Alt+File+Open selection to open a file, for example. However, the actual file-open dialog boxes were often quite dissimilar.

It wasn't until Windows 3.1 that the common dialog box library became part of the Windows API. Much of this library is exposed in the .NET Framework and consists of dialog boxes to open and close files, select colors and fonts, and aid in printing. The class hierarchy is as follows:

Both CommonDialog and FileDialog are abstract classes and can't be instantiated. That leaves six classes that you can use in applications. I'll cover ColorDialog, FontDialog, OpenFileDialog, and SaveFileDialog in this chapter, and PageSetupDialog and PrintDialog when I go deeper into printing in Chapter 21.

Choosing Fonts and Colors

Let's take a look at both FontDialog and ColorDialog in a single program that lets you set the BackColor, ForeColor, and Font properties of a form.

FontAndColorDialogs.cs
//----------------------------------------------------------------------------
// FontAndColorDialogs.cs © 2001 by Charles Petzold
//----------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FontAndColorDialogs:Form
{
    public static void Main()
    {
        Application.Run(new FontAndColorDialogs());
    }
    public FontAndColorDialogs()
    {
        Text = "Font and Color Dialogs";
        ResizeRedraw = true;
        Menu = new MainMenu();
        Menu.MenuItems.Add("&Format");
        Menu.MenuItems[0].MenuItems.Add("&Font...",
            new EventHandler(MenuFontOnClick));
        Menu.MenuItems[0].MenuItems.Add("&Background Color...",
            new EventHandler(MenuColorOnClick));
    }
    void MenuFontOnClick(object obj, EventArgs ea)
    {
        FontDialog fontdlg = new FontDialog();
        fontdlg.Font = Font;
        fontdlg.Color = ForeColor;
        fontdlg.ShowColor = true;
        if(fontdlg.ShowDialog() == DialogResult.OK)
        {
            Font = fontdlg.Font;
            ForeColor = fontdlg.Color;
            Invalidate();
        }
    }
    void MenuColorOnClick(object obj, EventArgs ea)
    {
        ColorDialog clrdlg = new ColorDialog();
        clrdlg.Color = BackColor;
        if (clrdlg.ShowDialog() == DialogResult.OK)
            BackColor = clrdlg.Color;
    }
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    StringFormat strfmt = new StringFormat();

    strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

    grfx.DrawString("Hello common dialog boxes!", Font,
                   new SolidBrush(ForeColor),
                   this.ClientRectangle, strfmt);
}

The program creates a Format menu containing two items: Font and Background Color. Although this program uses prewritten common dialog boxes rather than dialog boxes that we've written ourselves, the structure of the two menu Click event handlers should look very familiar. Look at MenuColorOnClick, for example. The dialog box is first created:

```
ColorDialog clrdlg = new ColorDialog();
```

It's then initialized by setting a property:

```
clrdlg.Color = BackColor;
```

And then the ShowDialog method is called:

```
if (clrdlg.ShowDialog() == DialogResult.OK)
   BackColor = clrdlg.Color;
```

If ShowDialog returns DialogResult.OK, the program uses information from the dialog box. Although this program uses the form's Font, ForeColor, and BackColor properties to initialize the two dialog boxes and then sets these properties based on what the user sets in the dialog boxes, you could instead define fields in your form that are associated with the dialog box settings.

The FontDialog class lets the user choose both a font and a font color identified by the following two properties:

```
FontDialog Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font</td>
<td>Font</td>
<td>get/set</td>
</tr>
<tr>
<td>Color</td>
<td>Color</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

The FontAndColorDialogs program simply initializes these two properties from the Font and ForeColor properties of the form and then sets these two form properties if the dialog box is terminated with the OK button.

The FontAndColorDialogs program sets the ShowColor property to enable the color option on the dialog box, which by default is disabled. The following properties let you control the appearance of the color option and other parts of the font dialog box:

```
FontDialog Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>ShowEffects</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>ShowColor</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>ShowApply</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>
```
If you set the `ShowEffects` property to `false`, the dialog box won't let you select underline or strikeout. Here's what the dialog box looks like when both `ShowEffects` and `ShowColor` are set to `true`:

![Font dialog box](image)

The `ShowApply` and `ShowHelp` options control the appearance of buttons labeled `Apply` and `Help`. If you enable these buttons, you'll want to handle the following events.

**FontDialog Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply</td>
<td>OnApply</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>HelpRequest</td>
<td>OnHelpRequest</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

We've already had experience in using an Apply button, but I'll also demonstrate using the Apply button in `FontDialog` in the next version of the program.

The `FontDialog` class has several other properties that control the appearance of fonts in the dialog box, but the defaults are usually sufficient. However, later on in this chapter (in the HeadDump program), I'll use the following property:

**FontDialog Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>FixedPitchOnly</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>

Set this property to `true` if you want to limit the display of fonts to those that have a uniform character width.

The `ColorDialog` dialog box lets you choose a color, which is represented by a property named `Color` of type `Color`.

**ColorDialog Properties (selection)**

| Type  | Property | Accessibility |
|-------|----------|---------------|-------------|
|       |          |               |             |
Several other bool properties control various aspects of the dialog box:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>FullOpen</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>AllowFullOpen</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>SolidColorOnly</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>AnyColor</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>ShowHelp</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>

The ColorDialog class doesn't support an Apply button. Here's the default color dialog box:

Normally, you just click one of the displayed colors and press OK. If none of those colors is satisfactory, you can click the button labeled Define Custom Colors. The dialog box expands like so:
What you can do now is define custom colors over at the right and then add them to the collection labeled Custom Colors at the left. You can't make the dialog box return to its original appearance unless you end it by pressing OK or Cancel. If you set the `FullOpen` property to `true` before calling `ShowDialog`, the dialog box opens in the wide version. If you set the `AllowFullOpen` property to `false` (regardless of the setting of `FullOpen`), the small version of the dialog box comes up and the Define Custom Colors button is disabled.

An application using the `ColorDialog` class has access to the custom colors through this property:

```
ColorDialog Properties (selection)
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int[]</code></td>
<td><code>CustomColors</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

The custom colors are stored as an array of sixteen 32-bit integers, where red is the least significant byte, green the next byte, blue the next, and the most significant byte is 0. (*Do not* use these integers in the static `Color.FromArgb` method that returns a `Color` object based on an integer. That method interprets blue as the least significant byte, and then green, red, and an alpha value where 0 is transparent.)

Suppose a user invokes the color dialog box from the FontAndColorDialogs program, clicks the Define Custom Colors button, and then carefully defines 16 custom colors. The user then selects one of them and presses OK. Then the user invokes the dialog box again and…. The custom colors are gone!

What happened? It's very simple. Look at the `MenuColorOnClick` method. The program re-creates the dialog box every time it's invoked. At the end of the `MenuColorOnClick` method, there are no more references to the `ColorDialog` object named `clrdlg`, so it becomes eligible for garbage collection.

If you leave the `AllowFullOpen` property set to `true`, it is very impolite not to save the custom colors between various invocations of the dialog box. You can do this in a couple ways. You can define an array of integers as a field:

```csharp
int[] aiCustomColors;
```

Before invoking the dialog box with `ShowDialog`, you set the property from the field, like so:

```csharp
clrdlg.CustomColors = aiCustomColors;
```

After `ShowDialog` returns, regardless of how the user ended the dialog box, the custom colors are saved back in the field:

```csharp
aiCustomColors = clrdlg.CustomColors;
```

There's an even easier approach. Simply remove the statement

```csharp
ColorDialog clrdlg = new ColorDialog();
```
from the `MenuColorOnClick` method and make it a field. Now the program uses only one instance of `ColorDialog` during the entire time the application is running.

Here's a better version of the `FontAndColorDialogs` program that also implements the Apply button in the font dialog box.

**BetterFontAndColorDialogs.cs**

```csharp
//--------------------------------------------------------
// BetterFontAndColorDialogs.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterFontAndColorDialogs:Form
{
    protected ColorDialog clrdlg = new ColorDialog();

    public static void Main()
    {
        Application.Run(new BetterFontAndColorDialogs());
    }
    public BetterFontAndColorDialogs()
    {
        Text = "Better Font and Color Dialogs";
        Menu = new MainMenu();
        Menu.MenuItems.Add("&Format");
        Menu.MenuItems[0].MenuItems.Add("&Font...",
        new EventHandler(MenuFontOnClick));
        Menu.MenuItems[0].MenuItems.Add("&Background Color...",
        new EventHandler(MenuColorOnClick));
        void MenuFontOnClick(object obj, EventArgs ea)
        {
            FontDialog fontdlg = new FontDialog();
            fontdlg.Font = Font;
            fontdlg.Color = ForeColor;
            fontdlg.ShowColor = true;
            fontdlg.ShowApply = true;
            fontdlg.Apply += new EventHandler(FontDialogOnApply);
            if(fontdlg.ShowDialog() == DialogResult.OK)
            {
                Font = fontdlg.Font;
            }
        }
        void MenuColorOnClick(object obj, EventArgs ea)
        {
        }
    }
}
```
ForeColor = fontdlg.Color;
Invalidate();
}
}
void MenuColorOnClick(object obj, EventArgs ea)
{
    clrdlg.Color = BackColor;

    if (clrdlg.ShowDialog() == DialogResult.OK)
        BackColor = clrdlg.Color;
}
void FontDialogOnApply(object obj, EventArgs ea)
{
    FontDialog fontdlg = (FontDialog) obj;

    Font = fontdlg.Font;
    ForeColor = fontdlg.Color;
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    grfx.DrawString("Hello common dialog boxes!", Font, new SolidBrush(ForeColor), 0, 0);
}

Now the custom colors are preserved when the dialog box is terminated andreshown.

Unfortunately, saving the dialog box settings from one invocation to another raises an additional question: How do you preserve settings when you terminate and rerun the program?

For that job, you probably want to take advantage of the Windows registry.

**Using the Windows Registry**

The Windows registry is a general-purpose mechanism that applications (and Windows itself) use to store program information that must be retained when an application terminates. The information is stored in a hierarchical format. You can use the Registry Editor program (Regedit.exe) that comes with Windows to examine (and, if you're very brave, even modify) the contents of the registry on your machine.

The information in the registry is organized by keys, which are often written in the syntax of directory paths. For example, in the Registry Editor, you can find the key HKEY_CURRENT_USER\Software\Microsoft\Notepad to examine all the information stored in the registry by the Microsoft Notepad program. Each piece of information has a name (for example, iPointSize and iWindowPosX), a type (in both these cases, a REG_DWORD, which is a 32-bit unsigned integer), and a value.

The Windows registry is supported with two classes in the Microsoft.Win32 namespace. The Registry class consists solely of seven static read-only fields for the seven possible root keys in the registry. The Description column shows how these root keys are defined in the Win32 header files and displayed in the Registry Editor:

<table>
<thead>
<tr>
<th>Registry Static Fields</th>
</tr>
</thead>
</table>
Most applications will probably restrict themselves to the **CurrentUser** key to store user-specific information such as favorite fonts, colors, and other settings.

The second class is **RegistryKey**. The following methods are probably the most common:

**RegistryKey Methods (selection)**

```
RegistryKey CreateSubKey(string strSubKey)
RegistryKey OpenSubKey(string strSubKey)
RegistryKey OpenSubKey(string strSubKey, bool bWritable)
void SetValue(string strName, object obj)
object GetValue(string strName)
void Close()
```

Notice that the `CreateSubKey` and `OpenSubKey` methods are members of the `RegistryKey` class and also return `RegistryKey` objects. The first `RegistryKey` object you obtain is from one of the fields of the `Registry` class, for example:

```
RegistryKey regkey = Registry.CurrentUser;
```

You then obtain another `RegistryKey` object by combining that registry key with a subkey argument passed to `CreateSubKey` or `OpenSubKey`. For example, if `regkey` has been obtained from `Registry.CurrentUser`, the call

```
regkey = regkey.OpenSubKey("Software\Microsoft\Notepad");
```

returns a registry key suitable for reading the information stored by Notepad. Or you can do both calls in one shot:

```
RegistryKey regkey =
    Registry.CurrentUser.OpenSubKey("Software\Microsoft\Notepad");
```

But that call obtains the key for Notepad. That's not the key you want to use for your application. You'll want to make your own key using the `CreateSubKey` method, for example:

```
RegistryKey regkey =
    Registry.CurrentUser.CreateSubKey("Software\MyCompany\MyApp");
```

You'll probably use `CreateSubKey` when your program is first installed or the first time it runs. Subsequently, you can use `OpenSubKey` to open the key for reading:

```
RegistryKey regkey =
    Registry.CurrentUser.OpenSubKey("Software\MyCompany\MyApp");
```

You can also use `OpenSubKey` to open the key for writing:

```
RegistryKey regkey =
```
After you're finished accessing the registry, close it like so:

```csharp
regkey.Close();
```

The `SetValue` and `GetValue` methods let you read and write values associated with names. But watch out: the syntax of the `SetValue` call makes it appear as if you can use an object of any type as the second argument, for example, an object of type `Font`:

```csharp
regkey.SetValue("MyFont", font);
```

This call will work (kind of), but the problem arises when you try to retrieve that same object with a call to `GetValue`:

```csharp
font = (Font) regkey.GetValue("MyFont");  // Won't work!
```

If the registry had been originally designed with an object-oriented interface in mind, these two calls might work. But it wasn't, and they won't. Basically, you're limited to storing strings, 32-bit integers, and byte arrays. (A byte array lets you store generalized binary information, though probably not as conveniently as you'd like.)

The `SetValue` call just shown actually stores `font.ToString()`, which is a string that describes the `Font` object. When you call `GetValue`, however, that string can't be cast into an object of type `Font`, and the invalid cast will cause a run-time exception. If you need to store an object of type `Font` in the registry, you must store everything you need to re-create the font in the form of strings, 32-bit integers, and byte arrays.

Let's look at an example. The `DialogsWithRegistry` class in the following program subclasses `BetterFontAndColorDialogs` and adds registry support. The seven `const` fields define the registry key and all the registry names I use in the program.

```csharp
// DialogsWithRegistry.cs
//--------------------------------------------------
// DialogsWithRegistry.cs © 2001 by Charles Petzold
// --------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class DialogsWithRegistry : BetterFontAndColorDialogs
{
    const string strRegKey =
    "Software\ProgrammingWindowsWithCSharp\DialogsWithRegistry";
    const string strFontFace = "FontFace";
    const string strFontSize = "FontSize";
    const string strFontStyle = "FontStyle";
    const string strForeColor = "ForeColor";
    const string strBackColor = "BackColor";
    const string strCustomClr = "CustomColor";

    public new static void Main()
    {
        Application.Run(new DialogsWithRegistry());
    }
}
```
public DialogsWithRegistry()
{
    Text = "Font and Color Dialogs with Registry";

    RegistryKey regkey = Registry.CurrentUser.OpenSubKey(strRegKey);
    if (regkey != null)
    {
        Font = new Font((string) regkey.GetValue(strFontFace),
                        float.Parse(
                                    (string) regkey.GetValue(strFontSize)),
                        (FontStyle) regkey.GetValue(strFontStyle));

        ForeColor = Color.FromArgb(
                        (int) regkey.GetValue(strForeColor));

        BackColor = Color.FromArgb(
                        (int) regkey.GetValue(strBackColor));

        int[] aiColors = new int[16];
        for (int i = 0; i < 16; i++)
            aiColors[i] = (int) regkey.GetValue(strCustomClr + i);

        clrdlg.CustomColors = aiColors;
    }
    regkey.Close();
}
}

protected override void OnClosed(EventArgs ea)
{
    RegistryKey regkey =
                        Registry.CurrentUser.OpenSubKey(strRegKey, true);
    if (regkey == null)
    {
        regkey = Registry.CurrentUser.CreateSubKey(strRegKey);
    }

    regkey.SetValue(strFontFace, Font.Name);
    regkey.SetValue(strFontSize, Font.SizeInPoints.ToString());
    regkey.SetValue(strFontStyle, (int) Font.Style);
    regkey.SetValue(strForeColor, ForeColor.ToArgb());
    regkey.SetValue(strBackColor, BackColor.ToArgb());

    for (int i = 0; i < 16; i++)
        regkey.SetValue(strCustomClr + i, clrdlg.CustomColors[i]);
Let's look at the override of the `OnClosed` method first. `OnClosed` is called when the form has been closed. That's a good time for the program to write information to the registry. If the `OpenSubKey` call returns `null`, the program must be running for the first time, so it calls `CreateSubKey` to create the registry key. Each `SetValue` call stores either an integer or a string in the registry. For the form's `Font` property, three values must be stored: the `Name`, `SizeInPoints`, and `Style` properties. The `SizeInPoints` property of `Font` is a `float`, so that value is converted to a string representation with `ToString`. The `ToArgb` method of the `Color` class converts `Color` objects into integers.

Also take note of the `SetValue` call in the `for` loop that's used to store the custom colors. The value name is `strCustomClr + i` which creates names of CustomColor0, CustomColor1, through CustomColor15.

The values are loaded from the registry in the program's constructor. The form's font is re-created using the face name, point size, and style values. The point size has to be converted from a string back to a `float` using the static `Parse` method of the `Single` class. The static `Color.FromArgb` method converts the stored integers back into `Color` objects.

Because implementing registry support requires working with two blocks of code, registry read and write code can be difficult to debug. The best approach is to get all the `SetValue` calls working first. Monitor your progress with the Registry Editor. (F5 refreshes the display.) The Registry Editor also lets you delete an entire key, so you can test how well your program re-creates the registry entries from scratch. When you get all the `SetValue` calls working, then code the `GetValue` calls.

### The Open File Dialog Box

Both `OpenFileDialog` and `SaveFileDialog` inherit from the abstract class `FileDialog`, which implements a number of properties common to both classes. Both `OpenFileDialog` and `SaveFileDialog` are sealed, meaning that you can't inherit from them.

Both `OpenFileDialog` and `SaveFileDialog` are primarily responsible for returning to your program a fully qualified filename that the user specifies either by selecting from a list box or by manually typing. Considering that this is file I/O we're speaking of here, however, using these dialog boxes can be more complex than using those involved with fonts or color.

Let's look at `OpenFileDialog` first. Three properties are connected with the retrieval of a filename or filenames from the dialog box:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td>Multiselect</td>
<td>get/set</td>
</tr>
<tr>
<td><code>string</code></td>
<td>FileName</td>
<td>get/set</td>
</tr>
<tr>
<td><code>string[]</code></td>
<td>FileNames</td>
<td>get</td>
</tr>
</tbody>
</table>

By default, `Multiselect` is `false`, indicating that the user can select only one file from the dialog box, in which case `FileName` indicates the selected file.

Armed with only this information, let's take a look at `HeadDump`, a hexadecimal dump program that displays only as much of the selected file as can fit in the client area. This program makes use of the static `ComposeLine` method from the `HexDump` program in Appendix A (which you should consult for information about files and streams as implemented in .NET).

```csharp
// HeadDump.cs
// HeadDump.cs © 2001 by Charles Petzold
```
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class HeadDump : Form
{
    const string strProgName = "Head Dump";
    string strFileName = "";

    public static void Main()
    {
        Application.Run(new HeadDump());
    }

    public HeadDump()
    {
        Text = strProgName;
        Font = new Font(FontFamily.GenericMonospace,
                         Font.SizeInPoints);

        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Open...",
                                      new EventHandler(MenuFileOpenOnClick));
        Menu.MenuItems.Add("&Format");
        Menu.MenuItems[1].MenuItems.Add("&Font...",
                                        new EventHandler(MenuFormatFontOnClick));
    }

    void MenuFileOpenOnClick(object obj, EventArgs ea)
    {
        OpenFileDialog dlg = new OpenFileDialog();

        if (dlg.ShowDialog() == DialogResult.OK)
        {
            strFileName = dlg.FileName;
            Text = strProgName + " - " + Path.GetFileName(strFileName);
            Invalidate();
        }
    }

    void MenuFormatFontOnClick(object obj, EventArgs ea)
    {
        FontDialog dlg = new FontDialog();

        dlg.Font = Font;

        dlg.Font = Font;
    }
}
dlg.FixedPitchOnly = true;

if (dlg.ShowDialog() == DialogResult.OK)
{
    Font = dlg.Font;
    Invalidate();
}
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Brush brush = new SolidBrush(ForeColor);
    FileStream fs;

    try
    {
        fs = new FileStream(strFileName, FileMode.Open,
                             FileAccess.Read, FileShare.Read);
    }
    catch
    {
        return;
    }

    for (int iLine = 0; iLine <= ClientSize.Height / Font.Height;
         iLine++)
    {
        byte[] abyBuffer = new byte[16];
        int iCount = fs.Read(abyBuffer, 0, 16);
        string str = HexDump.ComposeLine(16 * iLine,
                                         abyBuffer, iCount);

        grfx.DrawString(str, Font, brush, 0, iLine * Font.Height);
    }
    fs.Close();
}

Because hexadecimal dumps become a chaotic jumble when displayed with proportional fonts, this
form sets its Font property to the GenericMonospace font. The menu the program creates allows
changing this font to another fixed-pitch font as well as selecting a file.

The MenuFileOpenOnClick method shows how to create and display an OpenFileDialog object. If
ShowDialog returns DialogResult.OK, the program saves the FileName property in a field named
strFileName and then sets a new Text property for the form, using the static Path.GetFileName
method to extract the filename from the fully qualified path and filename. The OnPaint method is
responsible for opening the file, reading bytes, and formatting them. Here's the program displaying
its own source code file:
What you’ll find when experimenting with HeadDump is that the initial directory that opens in the open file dialog box is the one known as My Documents. You can then navigate to other directories. OpenFileDialog automatically saves the directory you finally select in the Windows registry. The next time you run the program, the open file dialog box will display the files in the last directory you navigated to using that dialog box.

By default, OpenFileDialog changes the current directory associated with the application as you’re navigating through the directories. If you want to set an initial directory for the dialog box or you want it to restore the current directory when the dialog box terminates, you can use the following two properties:

<table>
<thead>
<tr>
<th>FileDialog Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

Although most users navigate through directories and select files by picking the directories and files from lists, a user can also manually type a directory name or a filename. The following two properties are set to true by default so that the dialog box itself checks for valid path names and filenames before closing the dialog box:

<table>
<thead>
<tr>
<th>FileDialog Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

If you want to let the user create a new file using OpenFileDialog, set CheckFileExists to false.

The following two properties let you enable a check box labeled Open As Read-Only on the dialog box and determine whether the user checked it:

_OpenFileDialog Properties (selection) |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

If you enable this check box and the user checks it, your program shouldn’t save any changes to the file back to disk.

The other remaining properties I want to discuss involve the messy area of file types and filename extensions. You may have noticed that the OpenFileDialog in HeadDump has a combo box labeled Files Of Type that is blank:
For the HeadDump program, that's not so bad because the program can open and display any type of file. But for most programs, you want to force OpenFileDialog to display files of only specific types. You do this by using a text string called a filter. For the HeadDump program, I could have defined the filter like this:

"All Files (*.*)|*.*"

or like this:

"All Files|*.*"

The portion of the string up to the vertical bar is what the dialog box displays in the Files Of Type combo box. The string to the right of the vertical bar indicates the types of files the dialog box is to display, in this case, all files.

It's up to you whether or not the portion of the text string to the left of the vertical bar includes a file specification. That's the part the user sees, and it doesn't determine which files the dialog box displays.

If you were writing a clone of the Notepad program (which I'll actually be doing in Chapter 18), you would define the filter like so:

"Text Documents (*.txt)|*.txt|" +
"All Files|*.txt"

In this case, the Files Of Type combo box has two lines. When the user selects the first line in the combo box, the dialog box displays all files that have a txt extension; when the user selects the second line, the dialog box displays all files. Although I've written the string in two lines with a concatenation symbol (+) for clarity, it's really just one long string with the pieces separated by vertical bars:

"Text Documents (*.txt)|*.txt|All Files|*.txt"

There are always twice as many pieces of the string as there are lines in the combo box.

If you were writing a program that was able to load a variety of image files, you'd define a filter something like this:

"All Image Files|*.bmp;*.gif;*.jpeg;*.jpg;*.jfif;*.png;*.tif;*.tiff|" +
"Windows Bitmap (*.bmp)|*.bmp|" +
"Graphics Interchange Format (*.gif)|*.gif|" +
"JPEG File Interchange Format (*.jpg)|*.jpg;*.jpeg;*.jfif|" +
"Portable Network Graphics (*.png)|*.png|" +
"Tag Image File Format (*.tif)|*.tif;*.tiff|" +
"All Files (*.*)|*.*";
The Files Of Type combo box would have seven lines in this case. As you'll notice, some of the file
types are associated with multiple filename specifications. These are separated by semicolons. For
example, if the user selects the line in the combo box that reads

JPEG File Interchange Format (*.jpg)

the dialog box displays files with extensions jpg, jpeg, and jfif. The part that's displayed in the combo
box could alternatively be

JPEG File Interchange Format

or

JPEG/JFIF

or

JPEG File Interchange Format (*.jpg, *.jpeg, *.jfif)

Use whatever you think is best for the user.

You use the following two properties to specify the filter to be used for the Files Of Type combo type
and the line number of the filter that is to be initially displayed:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Filter</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>FilterIndex</td>
<td>get/set</td>
</tr>
</tbody>
</table>

On return from ShowDialog, FilterIndex indicates the index of the line selected by the user in the
Files Of Type combo box. These line numbers begin at 1.

Here's a program that implements an OpenFileDialog that has a filter to specify every type of file
supported by the static Image.FromFile method.

```csharp
// ImageOpen.cs
// ImageOpen.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
class ImageOpen : Form
{
    protected string strProgName;
    protected string strFileName;
    protected Image image;

    public static void Main()
    {
        Application.Run(new ImageOpen());
    }
    public ImageOpen()
    {
```
Text = strProgName = "Image Open";
ResizeRedraw = true;

Menu = new MainMenu();
Menu.MenuItems.Add("&File");
Menu.MenuItems[0].MenuItems.Add(new MenuItem("&Open...",
    new EventHandler(MenuFileOpenOnClick),
    Shortcut.CtrlO));
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
    OpenFileDialog dlg = new OpenFileDialog();

dlg.Filter = "All Image Files|*.bmp;*.ico;*.gif;*.jpeg;*.jpg;" +
    "*.jfif;*.png;*.tif;*.tiff;*.wmf;*.emf|" +
    "Windows Bitmap (*.bmp)|*.bmp|" +
    "Windows Icon (*.ico)|*.ico|" +
    "Graphics Interchange Format (*.gif)|*.gif|" +
    "JPEG File Interchange Format (*.jpg)|" +
    "*.jpg;*.jpeg;*.jfif|" +
    "Portable Network Graphics (*.png)|*.png|" +
    "Tag Image File Format (*.tif)|*.tif;*.tiff|" +
    "Windows Metafile (*.wmf)|*.wmf|" +
    "Enhanced Metafile (*.emf)|*.emf|" +
    "All Files (*.*)|*.*|";

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        try
        {
            image = Image.FromFile(dlg.FileName);
        }
        catch (Exception exc)
        {
            MessageBox.Show(exc.Message, strProgName);
            return;
        }
        strFileName = dlg.FileName;
        Text = strProgName + " - " + Path.GetFileName(strFileName);
        Invalidate();
    }
}
protected override void OnPaint(PaintEventArgs pea)
{
This program is structurally similar to the HeadDump program except that it attempts to load the image immediately after `ShowDialog` returns. Although the dialog box itself checks to make sure that the file exists, nothing prevents the user from specifying an existing file that's not an image file. If the file is not an image file (or the file is corrupted in some way), `Image.FromFile` throws an exception and the program displays a message box reporting the problem. The `OnPaint` method simply uses `DrawImage` to display the loaded image.

**The Save File Dialog Box**

Has it occurred to you yet that we’re on the verge of writing a program that can convert between various bitmap file formats using the standard Open and Save dialog boxes? It's a little tricky to carry it off—and the implementation isn't quite optimum—but that's my goal in the remainder of the chapter.

The `SaveFileDialog` class adds just two properties to the properties defined by `FileDialog`:

<table>
<thead>
<tr>
<th><code>SaveFileDialog Properties</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>bool</code></td>
</tr>
</tbody>
</table>

These two properties affect the display of message boxes that can be displayed while `SaveFileDialog` is still displayed. If you set `CreatePrompt` to `true` and the user specifies a file that doesn't exist, the dialog box will display a message box asking whether the user really wants to create that file. If you leave `OverwritePrompt` set to `true`, the dialog box asks for confirmation if the user selects a file that already exists.

Usually when an application invokes a dialog box to save a document, the application suggests a filename, and very often a default filename extension. For example, Notepad displays a filename of `*.txt` in the save file dialog box.

For a program that can convert between image formats, however, it's more proper for the save file dialog box to suggest a filename *without* an extension. The filename is the same as the filename of the loaded file. But the filename extension must be based on whatever format the user wants to save the file in, and that's not known when the dialog box is displayed.

Here's a way to do it. First, before displaying the dialog box, set the `FileName` property of `SaveFileDialog` to the opened filename without the extension. You can use the static `Path.GetFileNameWithoutExtension` method to strip an extension from a filename. Second, specify that the dialog box itself appends an extension to the selected file by setting the following property to `true`:

<table>
<thead>
<tr>
<th><code>FileDialog Properties (selection)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td><code>bool</code></td>
</tr>
</tbody>
</table>

The extension that `SaveFileDialog` appends to the filename is the first filename extension listed in the line of the filter that the user selects in the Save As Type combo box.

Here's one possible implementation. The `ImageIO` class overrides the `ImageOpen` class and adds a Save As item to the menu.

*ImageIO.cs*
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImageIO: ImageOpen
{
    MenuItem miSaveAs;

    public new static void Main()
    {
        Application.Run(new ImageIO());
    }

    public ImageIO()
    {
        Text = strProgName = "Image I/O";

        Menu.MenuItems[0].Popup += new EventHandler(MenuFileOnPopup);
        miSaveAs = new MenuItem("Save &As...");
        miSaveAs.Click += new EventHandler(MenuFileSaveAsOnClick);
        Menu.MenuItems[0].MenuItems.Add(miSaveAs);
    }

    void MenuFileOnPopup(object obj, EventArgs ea)
    {
        miSaveAs.Enabled = (image != null);
    }

    void MenuFileSaveAsOnClick(object obj, EventArgs ea)
    {
        SaveFileDialog savedlg = new SaveFileDialog();

        savedlg.InitialDirectory = Path.GetDirectoryName(strFileName);
        savedlg.FileName = Path.GetFileNameWithoutExtension(strFileName);
        savedlg.AddExtension = true;
        savedlg.Filter = "Windows Bitmap (*.bmp|*.bmp)" +
            "Graphics Interchange Format (*.gif|*.gif)" +
            "JPEG File Interchange Format (*.jpg)" +
            "*.jpg;*.jpeg;*.jfif" +
            "Portable Network Graphics (*.png|*.png)" +
            "Tagged Image File Format (*.tif|*.tif;*.tiff);";

        if (savedlg.ShowDialog() == DialogResult.OK)
try {
    image.Save(savedlg.FileName);
} catch (Exception exc) {
    MessageBox.Show(exc.Message, Text);
    return;
}
strFileName = savedlg.FileName;
Text = strProgName + " - " + Path.GetFileName(strFileName);
}

Notice that the filter for the `SaveFileDialog` object doesn't include an All Files or All Image Files line and includes only those formats that work with the `Save` method of the `Image` class. The idea here is that the user accepts the filename (without the extension) displayed in the dialog box and selects the format using the Save As Type combo box. If the user selects “JPEG File Interchange Format,” for example, the dialog box appends a .jpg filename extension to the base filename.

I mentioned at the beginning of this discussion that the implementation isn't optimum. That's because I think the absence of a displayed filename extension may be confusing to the more sophisticated user. It's not clear whether or not the program will append a filename extension. Fortunately, if the user types an extension, the dialog box won't append another one.

What's the best approach? I like an approach in which the filename is always displayed with an extension, and as the user selects different lines in the Save As Type combo box, the displayed filename extension changes accordingly. When you're programming using the Win32 API, it's possible to get access to the various controls in the common dialog boxes and perform little tricks like this. In the Windows Forms interface, however, access to those controls is hidden away.

Another approach to implementing format conversions is to force the user to make a decision before the save file dialog box is even displayed. You do this by making a submenu of the Save As menu item that lists the various formats: Windows Bitmap, Graphics Interchange Format, JPEG, and so forth. Each of these items invokes a `SaveFileDialog` that has a single-line filter for a specific file type.
Chapter 17: **Brushes and Pens**

**Overview**

Pens and brushes are fundamental to the Windows Forms graphics system. You use pens to draw straight lines and curves, and you use brushes to fill areas enclosed by straight lines and curves, and to draw text. It's hard to get started at all in Windows Forms without knowing something about pens and brushes. You can't even display text without knowing what a brush is.

Yet brushes are also one of the most all-encompassing objects in Windows Forms because you can create brushes based on paths and bitmaps, neither of which is an elementary graphics topic. Moreover, if brushes encompass almost everything we've learned about Windows Forms graphics so far, then pens are even more encompassing because pens can be based on brushes.

Here's the class hierarchy of the classes I'll be discussing in this chapter:

Both **Brush** and **Pen** are defined in the **System.Drawing** namespace, but **Brush** is an abstract class and hence can't be instantiated. Of the five classes derived from **Brush**, **SolidBrush** and **TextureBrush** are defined in the **System.Drawing** namespace, and **HatchBrush**, **LinearGradientBrush**, and **PathGradientBrush** are defined in the **System.Drawing.Drawing2D** namespace.

Because all five nonabstract brush classes are derived from **Brush**, it's often convenient to store an instance of one of these classes in a variable of type **Brush**. I've often done so in this book:

```csharp
Brush brush = new SolidBrush(ForeColor);
```

However, the classes derived from **Brush** define their own read/write properties, and it's easier to read and write these properties if you save the object in a variable of its own type.

**Filling in Solid Colors**

**SolidBrush** is by far the simplest of the five classes derived from **Brush**. The class has just one constructor, which we've been using since Chapter 3:

**SolidBrush Constructor**

```csharp
SolidBrush (Color color)
```

And the class has just one property:

**SolidBrush Property**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
</table>
In many cases, when you need a solid brush, you'll probably take advantage of the *Brushes* class, which has 141 static read-only properties, each of which returns a *Brush* object based on one of the standard colors.

**Hatch Brushes**

The hatch brush fills an area with a small repeating pattern, most commonly consisting of horizontal, vertical, or diagonal lines. At first, the hatch brush seems like one of the quainter approaches to computer graphics, conjuring up an ancient age of black-and-white bar graphs, pie charts, and other staples of business graphics.

While the use of color has certainly made hatch brushes less necessary, hatch brushes can continue to play a role in graphics output. Many users (myself included) still prefer noncolor laser printers to color ink jet printers. In some cases—when you're printing color photographs on a black-and-white printer, for example—different colors are mapped to gray shades and the results are often satisfactory. But sometimes different colors are used to represent data, for example, to indicate varying population levels on a map. Such colored graphics suffer greatly when the colors are blindly rendered as gray shades. Using hatch brushes instead allows an easier interpretation of the patterns on the map.

In Windows Forms, hatch patterns are based on monochrome 8-pixel-square bitmaps. Here's such a bitmap for a brick pattern:

This pattern doesn't begin looking like a brick until it's repeated horizontally and vertically:
The `HatchBrush` class is defined in the `System.Drawing.Drawing2D` namespace. There are only two constructors:

### HatchBrush Constructors

- `HatchBrush(HatchStyle hs, Color clrForeground)`
- `HatchBrush(HatchStyle hs, Color clrForeground, Color clrBackground)`

Although the bitmaps used for hatch brushes are monochrome, the 0's and 1's of the bitmap can be mapped to any two colors you want. In the simpler hatch brushes with line patterns, the foreground color is the color of the lines themselves; the background color is the color between the lines.

Watch out: The first time you use the first constructor, you'll probably choose black for the foreground color, like this:

```csharp
new HatchBrush(hs, Color.Black)   // Won't work right!
```

or this:

```csharp
new HatchBrush(hs, ForeColor)     // Probably won't work right!
```

The problem is that the default background color is black, and you should choose white for maximum brush contrast:

```csharp
new HatchBrush(hs, Color.White)   // Correct!
```

The more explicit equivalent is

```csharp
new HatchBrush(hs, Color.White, Color.Black)   // Correct!
```

Many hatch brushes will look fine with the foreground and background colors swapped like so:

```csharp
new HatchBrush(hs, Color.Black, Color.White)   // Satisfactory
```

However, with swapped colors, the brick pattern is rendered as white bricks and black mortar, which looks a bit peculiar.

A `HatchBrush` object is defined entirely by three properties, which are the same as the three arguments specified in the second constructor:

### HatchBrush Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>HatchStyle</td>
<td>HatchStyle</td>
<td>get</td>
</tr>
<tr>
<td>Color</td>
<td>ForegroundColor</td>
<td>get</td>
</tr>
<tr>
<td>Color</td>
<td>BackgroundColor</td>
<td>get</td>
</tr>
</tbody>
</table>

The `HatchStyle` property is a member of the `HatchStyle` enumeration. What makes Windows Forms GDI+ different from Windows GDI in the implementation of hatch brushes is the sheer number of hatch styles. GDI has 6 hatch styles (horizontal, vertical, two diagonal, and two cross hatches); GDI+ has 53 of them.

The `HatchStyle` enumeration provides members for all 53 styles (you'll see 54 in the documentation, but `Cross` and `LargeGrid` are identical), but it also has two members that ease the use of the enumeration values without hard-coding them:

### HatchStyle Enumeration (selection)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>4</td>
</tr>
</tbody>
</table>
In theory, what these members should tell you is that there are 53 hatch styles with values of 0 through 52. Unfortunately, HatchStyle.Max is inexplicably set to 4 rather than the expected 52. If HatchStyle.Max is ever corrected to reflect the true maximum value, it could be useful in generalizing code that presents all the possible hatch styles to the user. In the meantime, the two sample programs coming up contain hard-coded values for the minimum and maximum HatchStyle values.

The following program displays 32-pixel-square rectangles showing all the hatch styles.

```
HatchBrushArray.cs
//-------------------------------
// HatchBrushArray.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HatchBrushArray: PrintableForm
{
    const int iSize = 32, iMargin = 8;
    const int iMin = 0, iMax = 52;  // HatchStyle minimum and maximum values

    public new static void Main()
    {
        Application.Run(new HatchBrushArray());
    }
    public HatchBrushArray()
    {
        Text = "Hatch Brush Array";
        ClientSize = new Size(8 * iSize + 9 * iMargin,
                              7 * iSize + 8 * iMargin);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        for (HatchStyle hs = (HatchStyle)iMin; hs <= (HatchStyle)iMax; hs++)
        {
            HatchBrush hbrush =
                new HatchBrush(hs, Color.White, Color.Black);
            int y = (int)hs / 8;
            int x = (int)hs % 8;
            grfx.FillRectangle(hbrush, iMargin + x * (iMargin + iSize),
                               iMargin + y * (iMargin + iSize),
                               iSize, iSize);
        }
    }
}
```
The screen output looks like this:

![Hatch Brush Array Image]

Because the rectangles are 32 pixels square, each rectangle shows the 8-pixel-square hatch pattern repeated 16 times—4 times horizontally times 4 times vertically.

You can also print the hatch patterns by clicking the client area. Different printers might render the hatch patterns a little differently. For example, on a laser printer I have, each 8-pixel-square hatch pattern is displayed as a 1/15-inch square. Because the default page transform of the printer makes it appear to be a 100-dpi device, the 32-pixel-square rectangles displayed by HatchBrushArray are 0.32 inch square, a dimension that allows almost 25 repetitions of the pattern—5 times horizontally times 5 times vertically. When I direct the printer output to my fax machine, however, each rectangle shows 64 repetitions of the pattern.

Hatch patterns are not affected by transforms! If you insert statements in HatchBrushArray that set a nondefault page transform or world transform, you'll find that the transforms affect the location and size of the displayed rectangles (of course), but they don't affect the appearance of the pattern. The hatch lines don't spread wider or become closer; nor are they rotated or sheared.

In a drawing program, you may want to include a convenient way for a user to select a hatch style. Here's a program that puts all the hatch styles on a menu.

**HatchBrushMenu.cs**

```csharp
// HatchBrushMenu.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HatchBrushMenu : Form
{
    HatchStyleMenuItem hsmiChecked;
    //...
const int iMin = 0, iMax = 52;  // HatchStyle minimum and maximum values

public static void Main()
{
    Application.Run(new HatchBrushMenu());
}

public HatchBrushMenu()
{
    Text = "Hatch Brush Menu";
    ResizeRedraw = true;

    Menu = new MainMenu();
    Menu.MenuItems.Add("&Hatch-Style");

    for (HatchStyle hs = (HatchStyle)iMin; hs <= (HatchStyle)iMax; hs++)
    {
        HatchStyleMenuItem hsmi = new HatchStyleMenuItem();
        hsmi.HatchStyle = hs;
        hsmi.Click += new EventHandler(MenuHatchStyleOnClick);

        if ((int)hs % 8 == 0)
            hsmi.BarBreak = true;

        Menu.MenuItems[0].MenuItems.Add(hsmi);
    }
    hsmiChecked = (HatchStyleMenuItem) Menu.MenuItems[0].MenuItems[0];
    hsmiChecked.Checked = true;

} void MenuHatchStyleOnClick(object obj, EventArgs ea)
{
    hsmiChecked.Checked = false;
    hsmiChecked = (HatchStyleMenuItem) obj;
    hsmiChecked.Checked = true;

    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    HatchBrush hbrush = new HatchBrush(hsmiChecked.HatchStyle, Color.White, Color.Black);
The program defines a `HatchStyleMenuItem` class that subclasses `MenuItem` to provide a public `HatchStyle` field, but it also implements support for owner-draw items. When the application creates this menu, it inserts bar breaks every eight items. The resultant submenu is large, but not unwieldy, and provides a reasonable way for a user to select a hatch style:
It's about time we look at the actual names of these hatch styles. In the following tables, the images have a default background color of black and a foreground color of white. Each rectangle has 16 repetitions of the pattern (4 horizontally and 4 vertically).

The following six `HatchStyle` values are consistent with the Windows GDI:

**HatchStyle Enumeration (selection)**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal</td>
<td>0</td>
<td><img src="image1" alt="Image" /></td>
</tr>
<tr>
<td>Vertical</td>
<td>1</td>
<td><img src="image2" alt="Image" /></td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>2</td>
<td><img src="image3" alt="Image" /></td>
</tr>
<tr>
<td>BackwardDiagonal</td>
<td>3</td>
<td><img src="image4" alt="Image" /></td>
</tr>
<tr>
<td>Cross or LargeGrid</td>
<td>4</td>
<td><img src="image5" alt="Image" /></td>
</tr>
<tr>
<td>DiagonalCross</td>
<td>5</td>
<td><img src="image6" alt="Image" /></td>
</tr>
</tbody>
</table>

The following styles simulate gray shades. The member names indicate an approximate percentage of foreground color in the pattern:

**HatchStyle Enumeration (selection)**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent05</td>
<td>6</td>
<td><img src="image7" alt="Image" /></td>
</tr>
<tr>
<td>Percent10</td>
<td>7</td>
<td><img src="image8" alt="Image" /></td>
</tr>
<tr>
<td>Percent20</td>
<td>8</td>
<td><img src="image9" alt="Image" /></td>
</tr>
<tr>
<td>Percent25</td>
<td>9</td>
<td><img src="image10" alt="Image" /></td>
</tr>
<tr>
<td>Percent30</td>
<td>10</td>
<td><img src="image11" alt="Image" /></td>
</tr>
<tr>
<td>Percent40</td>
<td>11</td>
<td><img src="image12" alt="Image" /></td>
</tr>
<tr>
<td>Percent50</td>
<td>12</td>
<td><img src="image13" alt="Image" /></td>
</tr>
</tbody>
</table>
### HatchStyle Enumeration (selection)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent60</td>
<td>13</td>
<td><img src="percent60.png" alt="Image" /></td>
</tr>
<tr>
<td>Percent70</td>
<td>14</td>
<td><img src="percent70.png" alt="Image" /></td>
</tr>
<tr>
<td>Percent75</td>
<td>15</td>
<td><img src="percent75.png" alt="Image" /></td>
</tr>
<tr>
<td>Percent80</td>
<td>16</td>
<td><img src="percent80.png" alt="Image" /></td>
</tr>
<tr>
<td>Percent90</td>
<td>17</td>
<td><img src="percent90.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Next are some variations of the standard horizontal, vertical, and diagonal hatch styles:

### HatchStyle Enumeration (selection)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>LightDownwardDiagonal</td>
<td>18</td>
<td><img src="lightdownwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>LightUpwardDiagonal</td>
<td>19</td>
<td><img src="lightupwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>DarkDownwardDiagonal</td>
<td>20</td>
<td><img src="darkdownwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>DarkUpwardDiagonal</td>
<td>21</td>
<td><img src="darkupwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>WideDownwardDiagonal</td>
<td>22</td>
<td><img src="widedownwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>WideUpwardDiagonal</td>
<td>23</td>
<td><img src="wideupwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>LightVertical</td>
<td>24</td>
<td><img src="lightvertical.png" alt="Image" /></td>
</tr>
<tr>
<td>LightHorizontal</td>
<td>25</td>
<td><img src="lighthorizontal.png" alt="Image" /></td>
</tr>
<tr>
<td>NarrowVertical</td>
<td>26</td>
<td><img src="narrowvertical.png" alt="Image" /></td>
</tr>
<tr>
<td>NarrowHorizontal</td>
<td>27</td>
<td><img src="narrowhorizontal.png" alt="Image" /></td>
</tr>
<tr>
<td>DarkVertical</td>
<td>28</td>
<td><img src="darkvertical.png" alt="Image" /></td>
</tr>
<tr>
<td>DarkHorizontal</td>
<td>29</td>
<td><img src="darkhorizontal.png" alt="Image" /></td>
</tr>
<tr>
<td>DashedDownwardDiagonal</td>
<td>30</td>
<td><img src="dasheddownwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>DashedUpwardDiagonal</td>
<td>31</td>
<td><img src="dashedupwarddiagonal.png" alt="Image" /></td>
</tr>
<tr>
<td>DashedHorizontal</td>
<td>32</td>
<td><img src="dashedhorizontal.png" alt="Image" /></td>
</tr>
<tr>
<td>DashedVertical</td>
<td>33</td>
<td><img src="dashedvertical.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Finally, here's a group of miscellaneous patterns:

**HatchStyle Enumeration (selection)**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallConfetti</td>
<td>34</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>LargeConfetti</td>
<td>35</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>ZigZag</td>
<td>36</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Wave</td>
<td>37</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>DiagonalBrick</td>
<td>38</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>HorizontalBrick</td>
<td>39</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Weave</td>
<td>40</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Plaid</td>
<td>41</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Divot</td>
<td>42</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>DottedGrid</td>
<td>43</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>DottedDiamond</td>
<td>44</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Shingle</td>
<td>45</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Trellis</td>
<td>46</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>Sphere</td>
<td>47</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>SmallGrid</td>
<td>48</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>SmallCheckerBoard</td>
<td>49</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>LargeCheckerBoard</td>
<td>50</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>OutlinedDiamond</td>
<td>51</td>
<td><img src="url" alt="Image" /></td>
</tr>
<tr>
<td>SolidDiamond</td>
<td>52</td>
<td><img src="url" alt="Image" /></td>
</tr>
</tbody>
</table>

The Rendering Origin

There's something you should know about the hatch brush, and it's something that affects more sophisticated brushes as well. In fact, it's part of the model under which the Windows Forms graphics system was developed. To illustrate what I'm talking about, consider the following program.

**OverlappingHatchBrushes.cs**

```csharp
// OverlappingHatchBrushes.cs © 2001 by Charles Petzold
```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class OverlappingHatchBrushes: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new OverlappingHatchBrushes());
    }
    public OverlappingHatchBrushes()
    {
        Text = "Overlapping Hatch Brushes";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        HatchBrush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
            Color.White);
        for (int i = 0; i < 10; i++)
            grfx.FillRectangle(hbrush, i * cx / 10, i * cy / 10,
                cx / 8, cy / 8);
    }
}

This program uses the same hatch brush to draw 10 overlapping rectangles. But the rectangles don't seem to be distinct because the hatches in each rectangle coincide:

When you fill an area with a hatch pattern, the pattern is simply repeated horizontally and vertically. But that doesn't tell the whole story. The pattern has to be initially oriented at a particular pixel position. You might have suspected that the pattern is oriented with the graphics object being drawn,
for example, with the upper left corner of a rectangle. But this program pretty much demonstrates that's not the case. In fact, the hatch pattern bitmap is oriented with the upper left corner of the drawing area, which is the upper left corner of the client area or the printable area of the printer page.

A good way to think about this is that the brush blankets the entire display surface that your Graphics object refers to. When you draw a filled area with a particular brush, you're actually making a hole or a stencil that looks into that patterned surface.

You can override the default behavior of hatch-brush orientation by using the property of the Graphics class.

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
</tr>
</tbody>
</table>

The RenderingOrigin property affects only the display of hatch brushes. (Other types of brushes have other approaches to changing brush orientation.) By default, the RenderingOrigin property is the point (0, 0). If you set a new point in device coordinates, subsequent hatch brushes will be oriented with that point.

Here's a program that draws 10 staggered rectangles, using a different RenderingOrigin value for each.

```csharp
// HatchBrushRenderingOrigin.cs
//--------------------------------------------------------
// HatchBrushRenderingOrigin.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HatchBrushRenderingOrigin: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new HatchBrushRenderingOrigin());
    }
    public HatchBrushRenderingOrigin()
    {
        Text = "Hatch Brush Rendering Origin";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        HatchBrush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
                                              Color.White);
        for (int i = 0; i < 10; i++)
        {
            grfx.RenderingOrigin = new Point(i * cx / 10, i * cy / 10);
        }
    }
}
```
The result shows that the hatch pattern begins anew at the upper left corner of each rectangle:

![Image of hatch pattern]

There are a couple situations in which you’ll want to change the rendering origin. If you’re using hatch patterns for bar graphs, all the patterns will normally be oriented with the upper left corner of the drawing area. Even if you’re using a different pattern for each bar, the patterns can connect with each other in distracting ways. In such a case, you’ll want to set a new rendering origin for each bar based on the upper left corner of the bar.

You can also experience the opposite problem. Say you’re coloring a parent window and one or more child windows with the same hatch pattern. You probably want the child window to blend in with the parent. But the hatch pattern the child draws is oriented with the child’s upper left corner. The child will want to set the rendering origin to the upper left corner of its parent.

**Texture Brushes**

If you’ve gone through the list of hatch brushes and not discovered one you like, or if you need more than two colors in your brush, or if you want a brush that’s subject to transforms, you’ll want to explore the `TextureBrush` class.

A texture brush is based on an object of type `Image`—or a rectangular subset of an `Image` object—that repeats horizontally and vertically.

**TextureBrush Constructors**

```csharp
TextureBrush(Image image)
TextureBrush(Image image, Rectangle rectSrc)
TextureBrush(Image image, RectangleF rectfSrc)
TextureBrush(Image image, WrapMode wm)
TextureBrush(Image image, WrapMode wm, Rectangle rectSrc)
TextureBrush(Image image, WrapMode wm, RectangleF rectfSrc)
TextureBrush(Image image, Rectangle rectSrc, ImageAttributes ia)
TextureBrush(Image image, RectangleF rectSrc, ImageAttributes ia)
```
The `WrapMode` enumeration is defined in the `System.Drawing.Drawing2D` namespace:

**WrapMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile</td>
<td>0</td>
</tr>
<tr>
<td>TileFlipX</td>
<td>1</td>
</tr>
<tr>
<td>TileFlipY</td>
<td>2</td>
</tr>
<tr>
<td>TileFlipXY</td>
<td>3</td>
</tr>
<tr>
<td>Clamp</td>
<td>4</td>
</tr>
</tbody>
</table>

As we'll see shortly, the enumeration determines how the image repeats horizontally and vertically. The two versions of the constructor with an `ImageAttributes` argument don't require a `WrapMode` argument because `ImageAttributes` has its own method to set the wrap mode.

Here's a program that constructs a texture brush from a subset of the Apollo 11 image from Chapter 11. A menu option lets you select the wrap mode. The `OnPaint` method displays two overlapping ellipses using the brush.

**TextureBrushDemo.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TextureBrushDemo: PrintableForm
{
    MenuItem miChecked;
    TextureBrush tbrush;

    public new static void Main()
    {
        Application.Run(new TextureBrushDemo());
    }
    public TextureBrushDemo()
    {
        Text = "Texture Brush Demo";

        Image image = Image.FromFile("..\..\..\Images and Bitmaps\Apollo11FullColor.jpg");
        tbrush = new TextureBrush(image, new Rectangle(95, 0, 50, 55));

        Menu = new MainMenu();
    }
}
```
Menu.MenuItems.Add("&Wrap-Mode");

foreach (WrapMode wm in Enum.GetValues(typeof(WrapMode)))
{
    MenuItem mi = new MenuItem();
    mi.Text = wm.ToString();
    mi.Click += new EventHandler(MenuWrapModeOnClick);
    Menu.MenuItems[0].MenuItems.Add(mi);
}
miChecked = Menu.MenuItems[0].MenuItems[0];
miChecked.Checked = true;

void MenuWrapModeOnClick(object obj, EventArgs ea)
{
    miChecked.Checked = false;
    miChecked = (MenuItem) obj;
    miChecked.Checked = true;

    tbrush.WrapMode = (WrapMode)miChecked.Index;
    Invalidate();
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    grfx.FillEllipse(tbrush, 0, 0, 2 * cx / 3, 2 * cy / 3);
    grfx.FillEllipse(tbrush, cx / 3, cy / 3, 2 * cx / 3, 2 * cy / 3);
}

What this program does as well is demonstrate the see-through stencil effect of Windows Forms graphics. As long as the OnPaint method is using the same unaltered brush, the patterns will coincide exactly:
By default, WrapMode is *Tile*, which repeats the identical image horizontally and vertically. If you use the menu to change WrapMode to *TileFlipX*, the images in every other column are flipped left-to-right:

A WrapMode of *TileFlipY* turns the images in every other row upside down. The *TileFlipXY* option combines both effects.

The *Clamp* option results in the most extreme effect. The entire brush consists of only one rendition of the image in the upper left corner of the client area. The rest of the brush is transparent. You can see the bottom right part of the single image inside the ellipse:
As I mentioned earlier, the page transform and world transform do not affect hatch brushes. The transforms do affect texture brushes, however.

Suppose you modify the DoPage method of the TextureBrushDemo program so that it doesn't display an ellipse but instead displays a rectangle sized to fit exactly 9 repetitions of the 50 × 55 pixel image, for example:

```
grfx.FillRectangle(tbrush, 0, 0, 150, 165);
```

No matter what you do with the page transform or the world transform—regardless of any scaling, shearing, or rotation—the resultant parallelogram (for that, in general, is what the rectangle will be drawn as) will always be filled with exactly nine repetitions of the pattern, scaled, sheared, or rotated accordingly.

I also mentioned earlier that the RenderingOrigin property of the Graphics object affects only the HatchBrush object. The TextureBrush class itself has an alternative property that allows you to do something similar—plus lots more. Here's a complete list of the TextureBrush properties:

<table>
<thead>
<tr>
<th>TextureBrush Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>Image</td>
</tr>
<tr>
<td>WrapMode</td>
</tr>
<tr>
<td>Matrix</td>
</tr>
</tbody>
</table>

Yes, that third property is a matrix transform that affects the brush itself. For example, if you add the statement

```
tbrush.Transform = new Matrix(2, 0, 0, 2, 0, 0);
```

to the DoPage method of TextureBrushDemo, the ellipses (or rectangle or whatever you're drawing in that method) will be the same size and position, but the repeating image that makes up the pattern will be twice as large: 100 × 110 pixels. You can rotate or shear the pattern if you want. The TextureBrush class also includes the methods TranslateTransform, ScaleTransform, RotateTransform, MultiplyTransform, and ResetTransform, which are quite similar to the same-named methods in the Graphics class. (These are the only methods in TextureBrush that aren't in Brush.) Use translation to simulate a different rendering origin.

As with the similarly named methods in the Graphics class, the various transform methods in the TextureBrush class are cumulative. For example, if you put the statement

```
tbrush.RotateTransform(45);
```
in the `DoPage` method, the brush pattern will be rotated an additional 45 degrees whenever `DoPage` is called. To prevent unpredictable results, preface the `RotateTransform` call with a call to `ResetTransform`. Or better yet, put the `RotateTransform` call in the program's constructor. Here's what you'll get:

The `Transform` property of the `TextureBrush` class affects only the size and orientation of the brush pattern—not any objects you draw using the brush. The `Transform` property of the `Graphics` class affects both the size and orientation of the texture brush pattern and any objects you draw. If you use both, the pattern is affected by the composite transform.

**Linear Gradient Brushes**

The remaining two brush classes are `LinearGradientBrush` and `PathGradientBrush`; the word `gradient` here refers to a transition between colors. `LinearGradientBrush` involves a transition between two colors, sometimes called a fountain. At first, it may seem complicated to define a way in which one color merges with another, and that's probably why there are a couple different ways of specifying such a brush.

A gradient between two colors can be defined by a pair of parallel lines. Each line is a pure color. The color makes a transition between the two lines. Here's an example with the first color being `Color.LightGray` and the second `Color.DarkGray`:

The linear-gradient brush is thus an infinitely long stripe with two parallel borders of two colors.
To define such a brush, you don't need to specify two parallel lines. It's much easier to specify two points. The two parallel borders are at right angles to the line connecting the two points:

Note that there are an infinite number of pairs of points that result in the same linear gradient. I'm going to refer to the line connecting those two points as the gradient line. I also want to define the term mix line as the line at right angles to the gradient line and parallel to the two border lines.

The LinearGradientBrush class has eight constructors. Two of these let you specify two points and two colors:

**LinearGradientBrush Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinearGradientBrush(Point pt1, Point pt2, Color clr1, Color clr2)</td>
<td>Specifies two points and two colors.</td>
</tr>
<tr>
<td>LinearGradientBrush(PointF ptf1, PointF ptf2, Color clr1, Color clr2)</td>
<td>Specifies two points and two colors.</td>
</tr>
</tbody>
</table>

The only difference between these two constructors is the use of either Point or PointF structures. The points are in world coordinates. The color at the first point (pt1 or ptf1) is clr1, and the color at the second point (pt2 or ptf2) is clr2.

Let's take a look at a program that creates a LinearGradientBrush object in its DoPage method, defining the first point as $(cx/4, cy/4)$ and the second as $(3*cx/4, 3*cy/4)$. The two colors are Color.White and Color.Black. The program then colors a rectangle the size of its display area with this brush.

**TwoPointLinearGradientBrush.cs**

```csharp
// TwoPointLinearGradientBrush.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TwoPointLinearGradientBrush : PrintableForm
{
    public new static void Main()
```
Application.Run(new TwoPointLinearGradientBrush());
}

TwoPointLinearGradientBrush()
{
    Text = "Two-Point Linear Gradient Brush";
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    LinearGradientBrush lgbrush =
        new LinearGradientBrush(
            new Point(cx / 4, cy / 4),
            new Point(3 * cx / 4, 3 * cy / 4),
            Color.White, Color.Black);

    grfx.FillRectangle(lgbrush, 0, 0, cx, cy);
}

I haven't yet mentioned what happens outside the stripe that the LinearGradientBrush object defines. As you can see, by default the brush is tiled:

The wide continuous stripe from the lower left to the upper right is defined by the two brush coordinates. On either side of the stripe (in this case, the upper left and lower right of the client area), the stripe is repeated.

This behavior is controlled by the WrapMode property of the brush. WrapMode.Tile (the default) is the same as WrapMode.TileFlipY, and it causes the brush to be tiled with no flipping, as shown previously.

WrapMode.TileFlipX is the same as Wrapmode.TileFlipXY and causes the brush to be flipped so that there are no discontinuities, like so:
WrapMode.Clamp is not allowed for linear-gradient brushes.

Let me emphasize again that any Fill method you call with a particular brush essentially provides only a window through which you view the brush. When using texture brushes or gradient brushes, the appearance of any filled area depends to some degree on where the area is drawn. If you draw a small rectangle using the brush defined by the TwoPointLinearGradientBrush program, it might not even seem like much of a gradient.

In many cases, you’ll define a particular linear-gradient brush based on the actual coordinates of the object you’re filling. For example, if you want to fill a rectangle with a linear-gradient brush, you’ll define the brush with the same coordinates you use to draw the rectangle. In such cases, you might find it convenient to use the following constructors for LinearGradientBrush that have a rectangle argument:

**LinearGradientBrush Constructors (selection)**

```csharp
LinearGradientBrush(Rectangle rect, Color clr1, Color clr2,
                     LinearGradientMode lgm)
LinearGradientBrush(RectangleF rectf, Color clr1, Color clr2,
                     LinearGradientMode lgm)
```

The LinearGradientMode enumeration defines how the gradient line is formed from the sides or corners of the rectangle:

**LinearGradientMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal</td>
<td>0</td>
<td>Gradient line is horizontal, clr1 at left side, clr2 at right</td>
</tr>
<tr>
<td>Vertical</td>
<td>1</td>
<td>Gradient line is vertical, clr1 at top side, clr2 at bottom</td>
</tr>
<tr>
<td>ForwardDiagonal</td>
<td>2</td>
<td>Mix line passes through upper right and lower left corners; upper left corner is clr1 and lower right is clr2</td>
</tr>
<tr>
<td>BackwardDiagonal</td>
<td>3</td>
<td>Mix line passes through upper left and lower right corners; upper right corner is clr1 and lower left is clr2</td>
</tr>
</tbody>
</table>

Notice that for the last two enumeration values, two opposite corners of the rectangle define the mix line rather than the gradient line. Although the two other corners of the rectangle are pure colors and
the border lines pass through those two corners, those two corners do not define the gradient line unless the rectangle is a square. Let's take a closer look.

The following program defines a linear-gradient brush based on a rectangle that is half the width and height of the display area, and centered within the display area. You can use the menu to set the constructor's LinearGradientMode argument. After filling the display area with that brush, the DoPage method also draws the rectangle used in creating the brush.

```
RectangleLinearGradientBrush.cs
//-----------------------------------------------------------
// RectangleLinearGradientBrush.cs © 2001 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RectangleLinearGradientBrush : PrintableForm
{
    MenuItem miChecked;

    public new static void Main()
    {
        Application.Run(new RectangleLinearGradientBrush());
    }
    public RectangleLinearGradientBrush()
    {
        Text = "Rectangle Linear-Gradient Brush";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Gradient-Mode");

        foreach (LinearGradientMode gm in 
            Enum.GetValues(typeof(LinearGradientMode)))
        {
            MenuItem mi = new MenuItem();
            mi.Text = gm.ToString();
            mi.Click += new EventHandler(MenuGradientModeOnClick);
            Menu.MenuItems[0].MenuItems.Add(mi);
        }
        miChecked = Menu.MenuItems[0].MenuItems[0];
        miChecked.Checked = true;
    }
    void MenuGradientModeOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
    }
```
Invalidate();
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Rectangle rectBrush =
    new Rectangle(cx / 4, cy / 4, cx / 2, cy / 2);

    LinearGradientBrush lgbrush =
    new LinearGradientBrush(
        rectBrush, Color.White, Color.Black,
        (LinearGradientMode) miChecked.Index);

    grfx.FillRectangle(lgbrush, 0, 0, cx, cy);
    grfx.DrawRectangle(Pens.Black, rectBrush);
}

Here's an example when you've used the menu to select LinearGradientMode.ForwardDiagonal and the window has been widened somewhat:

![Gradient-Mode](image)

Although the upper left corner of the brush rectangle is colored with the first color and the lower right corner with the second color, the gradient line is obviously not the line from the upper left corner to the lower right corner because the gradient line is always at right angles to the border lines. Instead, the mix line (parallel to the border lines and midway between them) passes through the upper right and lower left corners of the rectangle.

The four final constructors for LinearGradientBrush let you specify a rectangle and an angle:

**LinearGradientBrush Constructors (selection)**

- LinearGradientBrush (Rectangle rect, Color clr1, Color clr2, float fAngle)
- LinearGradientBrush (Rectangle rect, Color clr1, Color clr2, float fAngle, bool bScale)
- LinearGradientBrush (RectangleF rectf, Color clr1, Color clr2, float fAngle)
- LinearGradientBrush (RectangleF rectf, Color clr1, Color clr2, float fAngle, bool bScale)
If \( fAngle \) is 0, the effect is identical to \textit{LinearGradientMode.Horizontal}: the gradient line is horizontal from the left side of the rectangle to the right side.

As \( fAngle \) increases, the gradient line rotates clockwise that number of degrees. The upper left corner of the rectangle is the first color, and the lower right corner is the second color. When \( fAngle \) reaches 90 degrees, the effect is identical to \textit{LinearGradientMode.Vertical}: the gradient line is vertical from the top of the rectangle to the bottom. As \( fAngle \) increases beyond 90 degrees, the gradient line continues to rotate clockwise. But now the upper right corner of the rectangle is the first color, and the lower left corner of the rectangle is the second color.

The optional \( bScale \) argument indicates whether the rotation angle is scaled by any transform associated with the brush.

I've already alluded to the \textit{WrapMode} and \textit{Transform} properties of \textit{LinearGradientBrush}. This table of four properties also includes the rectangle specified (or implied) by the constructor and an array of two colors used in the brush:

\[\begin{array}{|c|c|c|}
\hline
\textbf{Type} & \textbf{Property} & \textbf{Accessibility} \\
\hline
\text{RectangleF} & \text{Rectangle} & \text{get} \\
\hline
\text{Color[]} & \text{LinearColors} & \text{get/set} \\
\hline
\text{WrapMode} & \text{WrapMode} & \text{get/set} \\
\hline
\text{Matrix} & \text{Transform} & \text{get/set} \\
\hline
\end{array}\]

In addition to duplicating the \textit{TranslateTransform}, \textit{ScaleTransform}, \textit{RotateTransform}, \textit{MultiplyTransform}, and \textit{ResetTransform} methods defined in the \textit{TextureBrush} class, the \textit{LinearGradientBrush} class also includes these two methods:

\[\begin{array}{l}
\text{void SetBlendTriangularShape(float fFocus)} \\
\text{void SetBlendTriangularShape(float fFocus, float fScale)} \\
\text{void SetSigmaBellShape(float fFocus)} \\
\text{void SetSigmaBellShape(float fFocus, float fScale)} \\
\end{array}\]

Normally, the gradient is from the first color to the second color. These two methods change the gradient so that it goes from the first color to the second color and then back to the first. Both arguments (which I'll describe shortly) can range from 0 to 1.

Let's take a look. Here's the unaltered RectangleLinearGradientBrush program running with the default \textit{LinearGradientMode of Horizontal}:
The gradient is white at the left side of the rectangle and makes a transition to black at the right side of the rectangle. If you insert the statement

```
lgbrush.SetBlendTriangularShape(0.33f);
```

right after the brush creation statement, the output looks like this:

The gradient goes from white at the left side of the rectangle to black and then back to white at the right side of the rectangle. The `Focus` argument of 0.33 indicates that the black peaks at 1/3 of the way between the two sides of the rectangle.

When you replace that method call with

```
lgbrush.SetSigmaBellShape(0.33f);
```

the window looks like this:
The transition is more of a bell shape, again peaking 1/3 of the way between the left and right sides of the rectangle.

In both methods, the fScale argument indicates to what extent the gradient goes to the second color. The default is 1. Anything less than 1 causes the transition to go only partially to the second color. An fScale of 0 causes the brush to consist of only the first color, with no gradient.

To get even deeper into the control of the gradient colors, you can use the following three properties:

<table>
<thead>
<tr>
<th>LinearGradientBrush Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Blend</td>
</tr>
<tr>
<td>ColorBlend</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

Both **Blend** and **ColorBlend** are fairly simple classes. Both classes have just two read/write properties, which are both equally sized arrays. The two **Blend** properties are float arrays named **Positions** and **Factors**, which indicate the factors (0 through 1) used to scale the two colors at relative positions (ranging from 0 to 1) along the gradient line. The two **ColorBlend** properties are arrays named **Positions** and **Colors**, which indicate the colors at relative positions along the gradient line.

**Path Gradient Brushes**

The final type of brush is named **PathGradientBrush**. In the constructor, you define a polygon (which, as you know, is simply an array of points) and the brush is defined for the interior of the polygon. Alternatively, you can simply specify a **GraphicsPath** object:

**PathGradientBrush Constructors**

PathGradientBrush(Point[] apt)
PathGradientBrush(PointF[] aptf)
PathGradientBrush(Point[] apt, WrapMode wm)
PathGradientBrush(PointF[] aptf, WrapMode wm)
PathGradientBrush(GraphicsPath path)

Let's leap right into this subject by specifying a triangle in the **PathGradientBrush** constructor and seeing what happens.
TriagleGradientBrush.cs
//----------------------------------------------------
// TriangleGradientBrush.cs © 2001 by Charles Petzold
//----------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TriangleGradientBrush: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TriangleGradientBrush());
    }
    public TriangleGradientBrush()
    {
        Text = "Triangle Gradient Brush";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Point[] apt = { new Point(cx, 0),
                       new Point(cx, cy),
                       new Point(0, cy) };

        PathGradientBrush pgbrush = new PathGradientBrush(apt);

        grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
    }
}

Not much is happening in the program. An array of three points is defined (the upper right, lower right, and lower left corners of the display area), and then a PathGradientBrush is created from those points. The result, however, is quite cool:
Obviously, some default behavior was wisely defined!

Notice that I'm filling the entire client rectangle with this brush, yet the resultant object looks like a triangle. That's because I used a triangle to define the brush. Outside of the triangle, the brush is transparent. Whatever was on the display surface before the `FillRectangle` call will be unaffected.

In the generalized `PathGradientBrush`, gradients occur between each pair of points along the sides of the polygon and from the sides of the polygon to the center. The location of the center point and the colors are defined by the following three properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointF</td>
<td>CenterPoint</td>
<td>get/set</td>
</tr>
<tr>
<td>Color</td>
<td>CenterColor</td>
<td>get/set</td>
</tr>
<tr>
<td>Color[]</td>
<td>SurroundColors</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `CenterPoint` property is initially set to the average of the points in the polygon, which could actually be outside the polygon if the polygon has some concavity. The `CenterColor` property is initially set to `Color.Black`. The `SurroundColors` property is initialized as an array with one element equal to `Color.White`. You can set `SurroundColors` to an array of any size up to the number of points you used to create the brush.

For example, if you insert the line

```csharp
```

in the program right before the `FillRectangle` call, the point `(cx, 0)` will be red, `(cx, cy)` will be green, and `(0, cy)` will be blue. Try it!

Here's a program that creates a polygon describing a five-pointed star. It sets the center color to white and the surround color to black.

```csharp
// StarGradientBrush.cs
//------------------------------------------------
// StarGradientBrush.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
```
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class StarGradientBrush: PrintableForm
{
    public static void Main()
    {
        Application.Run(new StarGradientBrush());
    }

    public StarGradientBrush()
    {
        Text = "Star Gradient Brush";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Point[] apt = new Point[5];

        for (int i = 0; i < apt.Length; i++)
        {
            double dAngle = (i * 0.8 - 0.5) * Math.PI;
            apt[i] = new Point(
                (int)(cx * (0.50 + 0.48 * Math.Cos(dAngle)) ),
                (int)(cy * (0.50 + 0.48 * Math.Sin(dAngle)) ));
        }

        PathGradientBrush pgbrush = new PathGradientBrush(apt);

        pgbrush.CenterColor = Color.White;

        grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
    }
}

You can almost see how GDI+ draws the gradients as it circles around the points of the polygon. The latter gradients draw over the earlier ones and cause an effect that makes it look like part of the star goes through itself:
You can also use a path for defining a `PathGradientBrush` object. Here's a program that overrides the Bounce program from Chapter 11 to provide a new `DrawBall` method.

**BouncingGradientBrushBall.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class BouncingGradientBrushBall : Bounce
{
    public new static void Main()
    {
        Application.Run(new BouncingGradientBrushBall());
    }
    public BouncingGradientBrushBall()
    {
        Text = "Bouncing Gradient Brush Ball";
    }
    protected override void DrawBall(Graphics grfx, Rectangle rect)
    {
        GraphicsPath path = new GraphicsPath();
        path.AddEllipse(rect);

        PathGradientBrush pgbrush = new PathGradientBrush(path);
        pgbrush.CenterPoint = new PointF((rect.Left + rect.Right) / 3, (rect.Top + rect.Bottom) / 3);
        pgbrush.CenterColor = Color.White;
        pgbrush.SurroundColors = new Color[] { Color.Red };
As you'll recall, the DrawBall method is responsible for drawing a ball on a bitmap. The earlier version just drew a red Ellipse object using the Rectangle argument to the method. This version defines a path based on that ellipse and then creates a PathGradientBrush object based on that path. Normally, the gradient center would be the center of the ellipse, but this method moves the center a bit to the upper left. The center color is set to white, and the surround color is set to red. The method concludes by drawing a rectangle using this brush. (The brush is transparent outside the boundaries of the ellipse.) The resultant ball looks more realistic than the earlier one because the white spot gives the appearance of reflected light.

Tiling the Brush

Here are two other useful properties of PathGradientBrush:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>RectangleF</td>
<td>Rectangle</td>
<td>get</td>
</tr>
<tr>
<td>WrapMode</td>
<td>WrapMode</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Rectangle is a read-only property calculated by the brush when the brush is created. It is the smallest rectangle that encloses the polygon. This rectangle is not affected by the CenterPoint property; that is, CenterPoint is not necessarily inside this rectangle.

For a path-gradient brush, the WrapMode property is WrapMode.Clamp by default. Besides setting the WrapMode property, you can also optionally specify a nondefault value in the constructor. The reason I mention both the Rectangle and WrapMode properties together is because the effect of WrapMode is highly dependent on the rectangle.

Let's make another triangle, this one of a fixed size and occupying the upper left half of a square. A two-argument version of the constructor is used to set the wrap mode, which is based on a menu selection.

TriangleTile.cs

//-------------------------------------------
// TriangleTile.cs © 2001 by Charles Petzold

grfx.FillRectangle(pgbrush, rect);
}
}
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TriangleTile : PrintableForm
{
    const int iSide = 50;         // Side of square for triangle
    MenuItem miChecked;

    public new static void Main()
    {
        Application.Run(new TriangleTile());
    }
    public TriangleTile()
    {
        Text = "Triangle Tile";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Wrap-Mode");

        foreach (WrapMode wm in Enum.GetValues(typeof(WrapMode)))
        {
            MenuItem mi = new MenuItem();
            mi.Text = wm.ToString();
            mi.Click += new EventHandler(MenuWrapModeOnClick);
            Menu.MenuItems[0].MenuItems.Add(mi);
        }
        miChecked = Menu.MenuItems[0].MenuItems[0];
        miChecked.Checked = true;
    }
    void MenuWrapModeOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
        Invalidate();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Point[] apt = {
            new Point(0, 0),
            new Point(iSide, 0),
            new Point(0, iSide)};
        }
PathGradientBrush pgbrush =
    new PathGradientBrush(apt, (WrapMode) miChecked.Index);

    grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
}
}

Without the second argument to the constructor, the default WrapMode is WrapMode.Clamp, which means that the polygon isn't repeated at all. With WrapMode.Tile, which we've set as the initial wrap mode, the polygon is repeated horizontally and vertically over the entire filled area (in this case, the client rectangle).

You can, of course, achieve different kinds of effects by using different wrap modes. If you use WrapMode.TileFlipX, every other polygon is flipped around the vertical axis:
Similarly, `WrapMode.TileFlipY` causes every other polygon to be flipped around the horizontal axis:

Finally, the `WrapMode.TileFlipXY` option is a combination of the two effects:
And this is now beginning to look like an interesting pattern.

The uncolored white areas in that last screen shot are the background of the window showing through the transparent areas of the brush. The brush is only a triangle, so it can't entirely fill an area with tiling. Only a brush composed of a rectangle can fill an area with horizontal and vertical tiling, as in this program.

**SquareTile.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
class SquareTile: PrintableForm
{
    const int iSide = 50; // Side of square

    public new static void Main()
    {
        Application.Run(new SquareTile());
    }
    public SquareTile()
    {
        Text = "Square Tile";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Point[] apt = {new Point(0, 0), new Point(iSide, 0),
                      new Point(iSide, iSide), new Point(0, iSide)};
```
PathGradientBrush pgbrush =
    new PathGradientBrush/apt, WrapMode.TileFlipXY);  

pgbrush.SurroundColors = new Color[]{ Color.Red, Color.Lime,
    Color.Blue, Color.White};

grfx.FillRectangle(pgbrush, 0, 0, cx, cy);
}
)
)

This program looks pretty cool in color, even though this monochrome rendition doesn’t capture the full effect:

![Square Tile](image1)

But if you remove the SurroundColors assignment, the gray-shaded version also looks like an interesting pattern—or at least more interesting than the few statements of code would seem to imply:

![Square Tile](image2)
Although a rectangular brush is the only brush shape capable of filling an entire area with tiling, it's also possible to fill an entire area by using two (or more) nonrectangular brushes. Here's a program that creates two interlocking triangular brushes and fills the client area twice.

TwoTriangleTile.cs

class TwoTriangleTile: PrintableForm
{
    const int iSide = 50;         // Side of square for triangle
    public new static void Main()
    {
        Application.Run(new TwoTriangleTile());
    }
    public TwoTriangleTile()
    {
        Text = "Two-Triangle Tile";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        // Define the triangle and create the first brush.
        Point[] apt =
            {new Point(0, 0), new Point(iSide, 0), new Point(0, iSide)};
        PathGradientBrush pgbrush1 =
            new PathGradientBrush(apt, WrapMode.TileFlipXY);

        // Define another triangle and create the second brush.
        apt = new Point[] {new Point(iSide, 0), new Point(iSide, iSide),
            new Point(0, iSide)};
        PathGradientBrush pgbrush2 =
            new PathGradientBrush(apt, WrapMode.TileFlipXY);
        grfx.FillRectangle(pgbrush1, 0, 0, cx, cy);
        grfx.FillRectangle(pgbrush2, 0, 0, cx, cy);
Notice that the second polygon simply defines a triangle in the lower right corner of the square. The combination of the two triangle brushes fills the entire area:

Because the overall effect appears to be tiled squares, you might ask, Isn't it possible to do this pattern with one PathGradientBrush? No, it's not, because a PathGradientBrush has only one center. Each square in this pattern has two centers. (Of course, you could simulate this effect with a TextureBrush because then you're defining a tiled bitmap.)

Using PathGradientBrush is so much fun, it's hard to stop. Let's consider the following honeycomb-like pattern:

The black centers indicate that the polygon used in the PathGradientBrush object is a hexagon. Yet the tiling doesn't look possible. Each column of hexagons might be vertically tiled, but they certainly aren't horizontally tiled. The trick here, again, is to use two brushes. One brush does all the even columns of hexagons, and the other does the odd columns. Both brushes are tiled both horizontally and vertically.

HexagonGradientBrush.cs

//---------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HexagonGradientBrush: PrintableForm
{
    const float fSide = 50;       // Side (also radius) of hexagon

    public new static void Main()
    {
        Application.Run(new HexagonGradientBrush());
    }

    public HexagonGradientBrush()
    {
        Text = "Hexagon Gradient Brush";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        // Calculate half the hexagon height.
        float fHalf = fSide * (float) Math.Sin(Math.PI / 3);

        // Define a hexagon including some extra width.
        PointF[] aptf = { new PointF( fSide, 0),
                           new PointF( fSide * 1.5f, 0),
                           new PointF( fSide, 0),
                           new PointF( fSide / 2, -fHalf),
                           new PointF(-fSide / 2, -fHalf),
                           new PointF(-fSide, 0),
                           new PointF(-fSide * 1.5f, 0),
                           new PointF(-fSide, 0),
                           new PointF(-fSide / 2, fHalf),
                           new PointF( fSide / 2, fHalf) };

        // Create the first brush.
        PathGradientBrush pgbrush1 =
            new PathGradientBrush(aptf, WrapMode.Tile);

        // Offset the hexagon and define the second brush.
for (int i = 0; i < aptf.Length; i++)
{
    aptf[i].X += fSide * 1.5f;
    aptf[i].Y += fHalf;
}
PathGradientBrush pgbrush2 =
    new PathGradientBrush(aptf, WrapMode.Tile);
grfx.FillRectangle(pgbrush1, 0, 0, cx, cy);
grfx.FillRectangle(pgbrush2, 0, 0, cx, cy);
}

Pens Can Be Brushes Too

So far in this chapter I've been discussing brushes, but I've also been talking about pens, and that's because pens can be created from brushes. Here's the complete list of Pen constructors:

**Pen Constructors**

- Pen (Color clr)
- Pen (Color clr, float fWidth)
- Pen (Brush brush)
- Pen (Brush brush, float fWidth)

Creating a Pen from a SolidBrush object is equivalent to creating a Pen from the Color object that the SolidBrush object is based on.

With pens, it's very helpful to remember the stenciling effect that I mentioned earlier. When you draw with a pen created from a brush, you are effectively creating a slit through which you can view the brush. For example, here's a program that creates a Pen object based on a LinearGradientBrush.

**GradientPen.cs**

```csharp
//------------------------------------------
// GradientPen.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class GradientPen: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new GradientPen());
    }
    public GradientPen()
    {
        Text = "Gradient Pen";
    }
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Brush lgbrush = new LinearGradientBrush(
        new Rectangle(0, 0, cx, cy),
        Color.White, Color.Black,
        LinearGradientMode.BackwardDiagonal);

    Pen pen = new Pen(lgbrush, Math.Min(cx, cy) / 25);
    pen.Alignment = PenAlignment.Inset;
    grfx.DrawRectangle(pen, 0, 0, cx, cy);
    grfx.DrawLine(pen, 0, 0, cx, cy);
    grfx.DrawLine(pen, 0, cy, cx, 0);
}

The brush is based on a rectangle that encompasses the entire drawing area. The 
LinearGradientMode is set as BackwardDiagonal, which means that the mix line is from the upper 
left corner of the rectangle to the lower right corner. When you draw a line coinciding with (or parallel 
to) the mix line using a pen created with this brush, it has a constant color, not a gradient.

Although all the lines drawn by this program use the same pen, they have different gradients. The 
diagonal line from the lower left to the upper right goes from black to white. The horizontal and 
vertical lines go from black to gray, or gray to white.

In this program, I use a pen width that is a minimum of 1/25 of the width and height of the display 
area. The following table shows some width-related properties of the Pen class:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Width</td>
<td>get/set</td>
</tr>
<tr>
<td>Matrix</td>
<td>Transform</td>
<td>get/set</td>
</tr>
</tbody>
</table>
Pen Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PenAlignment</td>
<td>Alignment</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The Width property is in world coordinates, but it's never smaller than 1 pixel. If you specifically want a 1-pixel pen, specify a width of 0.

Along with the Transform property, the Pen class has the customary array of transform-setting methods: TranslateTransform, ScaleTransform, RotateTransform, MultiplyTransform, and ResetTransform. However, the transform does not affect the location and orientation of lines you draw with a pen, nor the brush that the pen may be based on. The transform affects only the pen width. The type of transform that makes most sense for pens is scaling. With ScaleTransform (or setting the Transform property manually), you can make pens that have different horizontal and vertical widths. For example, suppose you have a pen that is 10 units wide. If you call

```csharp
pen.ScaleTransform(2, 4);
```

the pen will have a horizontal width of 20 and a vertical width of 40, which means that vertical lines with this pen will have a width of 20 units, horizontal lines will have a width of 40 units, and diagonal lines will have a width somewhere in between. RotateTransform will rotate that effect.

I set the pen's Alignment property in the GradientPen program. The Alignment property can have one of the values of the PenAlignment enumeration defined in System.Drawing.Drawing2D.

PenAlignment Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td>0</td>
</tr>
<tr>
<td>Inset</td>
<td>1</td>
</tr>
<tr>
<td>Outset</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>3</td>
</tr>
<tr>
<td>Right</td>
<td>4</td>
</tr>
</tbody>
</table>

The Alignment property governs how wide pens appear when you draw rectangles or ellipses. By default, the property is PenAlignment.Center, which means that the wide pen is centered over the specified coordinates. In the GradientPen program, the lines drawn by DrawRectangle would be half outside the client area. Switching to PenAlignment.Inset causes the whole pen to appear inside the rectangle.

The following three properties concern the Brush or Color object used in the Pen:

Pen Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PenType</td>
<td>PenType</td>
<td>get</td>
</tr>
<tr>
<td>Color</td>
<td>Color</td>
<td>get/set</td>
</tr>
<tr>
<td>Brush</td>
<td>Brush</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The PenType enumeration is defined in System.Drawing.Drawing2D.

PenType Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolidColor</td>
<td>0</td>
</tr>
<tr>
<td>HatchFill</td>
<td>1</td>
</tr>
<tr>
<td>TextureFill</td>
<td>2</td>
</tr>
</tbody>
</table>
**PenType Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathGradient</td>
<td>3</td>
</tr>
<tr>
<td>LinearGradient</td>
<td>4</td>
</tr>
</tbody>
</table>

If you've created a pen with a color or with `SolidBrush`, the `Color` property of `Pen` is valid; otherwise, the `Brush` property is valid. However, you can change either the `Color` or `Brush` property of an existing pen and effectively change the pen type.

**A Dash of Style**

Pens needn't be solid lines. They can instead consist of patterns of dots and dashes. This style is specified with the pen property named `DashStyle`:

**Pen Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>DashStyle</td>
<td>DashStyle</td>
<td>get/set</td>
</tr>
<tr>
<td>float</td>
<td>DashOffset</td>
<td>get/set</td>
</tr>
<tr>
<td>float[]</td>
<td>DashPattern</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `DashStyle` enumeration is defined in `System.Drawing.Drawing2D`.

**DashStyle Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid</td>
<td>0</td>
</tr>
<tr>
<td>Dash</td>
<td>1</td>
</tr>
<tr>
<td>Dot</td>
<td>2</td>
</tr>
<tr>
<td>DashDot</td>
<td>3</td>
</tr>
<tr>
<td>DashDotDot</td>
<td>4</td>
</tr>
<tr>
<td>Custom</td>
<td>5</td>
</tr>
</tbody>
</table>

The appearance of the dots and dashes is affected by the pen width and any transforms that are in effect. The following program demonstrates this.

**PenDashStyles.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PenDashStyles : PrintableForm
{
    MenuItem miChecked;

    public new static void Main()
    {
```
The program constructs a menu that lets you select various widths from 1 through 25. The program displays five lines using the first five dash styles equally spaced in the client area. Here's the Morse code effect when you select a width of 25:
As I'll demonstrate in the next section, you have some control over the appearance of large dashes and dots.

If you need to draw a dashed or dotted polyline, use DrawLines or a path. Don't use multiple calls to DrawLine because the dash pattern starts anew with each line.

You can control how the dash begins in each line by using the DashOffset property. The property indicates an offset into the dash style where the pattern of dots and dashes begins. The offset is in increments of the dot size and is independent of the pen width. For example, if you insert the line

```csharp
pen.DashOffset = 1;
```

into the PenDashStyles program, the patterns begin one dot size later and look like this:

The DashOffset value is a float, so it can take on nonintegral values. If you want the DashDot or DashDotDot styles to begin with dots rather than dashes, set DashOffset equal to 4:

```csharp
pen.DashOffset = 4;
```

You can also set your own pattern of dots and dashes using the DashPattern property. The array of float values you specify indicates an alternating series of dash lengths and space lengths, all in increments of the dot size. Here's an example:

```csharp
float[] afDash = {2, 1, 4, 3};
p.pen.DashPattern = afDash;
```

After setting DashPattern, DashStyle is Custom and the line drawn with pen will contain a two-dot dash, one space, a four-dot dash, and three spaces.

### Caps and Joins

As the PenDashStyle program indicates, when lines start to get wide, they assume a graphical form of their own. You may like the square and rectangular appearance of the dots and dashes in styled lines, but you may prefer a more rounded appearance instead.

This is the realm of line caps (also known as ends) and joins. The cap governs the appearance of the lines at their beginning and end, or the appearance of the dots and dashes. The join governs what happens at the meeting of two connected lines. Here are the four basic caps and joins properties:

<table>
<thead>
<tr>
<th>Pen Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>LineCap</td>
</tr>
<tr>
<td>LineCap</td>
</tr>
</tbody>
</table>
I want to begin with the `LineJoin` property because that's probably the simplest. The property can take on one of the following enumeration values:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miter</td>
<td>0</td>
<td>Default, pointed</td>
</tr>
<tr>
<td>Bevel</td>
<td>1</td>
<td>Leveled off</td>
</tr>
<tr>
<td>Round</td>
<td>2</td>
<td>Rounded</td>
</tr>
<tr>
<td>MiterClipped</td>
<td>3</td>
<td>Pointed with limitations</td>
</tr>
</tbody>
</table>

The `LineJoin` property affects only lines that are connected, that is, polylines drawn with `DrawLines` or connected lines in a path. Here's a program that draws simple V-shaped polylines with the four different `LineJoin` values.

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class LineJoins: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new LineJoins());
    }
    public LineJoins()
    {
        Text = "Line Joins: Miter, Bevel, Round, MiterClipped";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen     penNarrow = new Pen(clr);
        Pen     penWide   = new Pen(Color.Gray, cx / 16);
        Point[] apt       = { new Point(1 * cx / 32, 1 * cy / 8),
                              new Point(4 * cx / 32, 6 * cy / 8),
                              new Point(7 * cx / 32, 1 * cy / 8) };
```
for (int i = 0; i < 4; i++)
{
    penWide.LineJoin = (LineJoin) i;
    grfx.DrawLine(penWide, apt);
    grfx.DrawLine(penNarrow, apt);
    grfx.TranslateTransform(cx / 4, 0);
}

And here's what it looks like:

The wide gray pen is the one whose LineJoin property is set. The thin black line shows the actual geometric line. You'll notice that the MiterClipped join looks just like Miter, but try making the form very tall: the Miter join continues to get longer and pointier, but at some point the MiterClipped join is truncated to look the same as a Bevel join. There's a reason to limit the length of miter joins: as the angle between two joined lines increases, the miter join can become very long. For example, a 1-inch-thick polyline joined at an angle of 1 degree would have a miter join that extended over 4½ feet! The Pen class has a special property to limit this extent when the LineJoin property is MiterClipped:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>MiterLimit</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The property truncates the miter join at a distance of $\text{pen.MiterLimit} \times \text{pen.Width} / 2$. The default MiterLimit is 10. If the Width property of the pen is 20, the miter extends only 100 units past the theoretical end of the line.

Let's now take a look at the DashCap property that affects the appearance of dots and dashes in styled lines. The property can take on one of the following enumeration values.

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td>0</td>
</tr>
<tr>
<td>Round</td>
<td>2</td>
</tr>
<tr>
<td>Triangle</td>
<td>3</td>
</tr>
</tbody>
</table>

Here's a variation of the PenDashStyles program that displays a DashDotDot line using the three different DashCap values.

PenDashCaps.cs

//---
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class PenDashCaps: PrintableForm
{
    MenuItem miChecked;

    public new static void Main()
    {
        Application.Run(new PenDashCaps());
    }
    public PenDashCaps()
    {
        Text = "Pen Dash Caps: Flat, Round, Triangle";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&Width");

        int[] aiWidth = { 1, 2, 5, 10, 15, 20, 25 };

        foreach (int iWidth in aiWidth)
            Menu.MenuItems[0].MenuItems.Add(iWidth.ToString(),
                new EventHandler(MenuWidthOnClick));

        miChecked = Menu.MenuItems[0].MenuItems[0];
        miChecked.Checked = true;
    }

    void MenuWidthOnClick(object obj, EventArgs ea)
    {
        miChecked.Checked = false;
        miChecked = (MenuItem) obj;
        miChecked.Checked = true;
        Invalidate();
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr, Convert.ToInt32(miChecked.Text));
        pen.DashStyle = DashStyle.DashDotDot;

        foreach (DashCap dc in Enum.GetValues(typeof(DashCap)))
        {
Here's the display when you select a width of 25:

These look a little odd because the beginning and end of the actual line is still squared off. The appearance of the beginning and end of the line is affected by the StartCap and EndCap properties, both of which are of type LineCap. You can insert the following statement into the PenDashCaps program to make these two properties consistent with the DashCap property:

```csharp
pen.StartCap = pen.EndCap = (LineCap) (int) dc;
```

The display then looks like this:

The two lines with the round and triangle caps aren't quite aligned with the flat caps. The reason they're not is that (as we'll see) the round and triangle caps go beyond the geometric point marking the beginning and end of the line. But the full width of the dashes and dots is kept consistent regardless of the cap style.

Here's the complete LineCap enumeration:

<table>
<thead>
<tr>
<th>LineCap Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Value</td>
</tr>
</tbody>
</table>


**LineCap Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td>0x00</td>
</tr>
<tr>
<td>Square</td>
<td>0x01</td>
</tr>
<tr>
<td>Round</td>
<td>0x02</td>
</tr>
<tr>
<td>Triangle</td>
<td>0x03</td>
</tr>
<tr>
<td>NoAnchor</td>
<td>0x10</td>
</tr>
<tr>
<td>SquareAnchor</td>
<td>0x11</td>
</tr>
<tr>
<td>RoundAnchor</td>
<td>0x12</td>
</tr>
<tr>
<td>DiamondAnchor</td>
<td>0x13</td>
</tr>
<tr>
<td>ArrowAnchor</td>
<td>0x14</td>
</tr>
<tr>
<td>AnchorMask</td>
<td>0xF0</td>
</tr>
<tr>
<td>Custom</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

And here's a program that draws wide lines using all these values. The width of the line is fixed at the `Font.Height` property. In addition, the program draws thin lines showing the geometric beginning and end of each line.

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class LineCaps : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new LineCaps());
    }
    public LineCaps()
    {
        Text = "Line Caps";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen penWide = new Pen(Color.Gray, Font.Height);
        Pen penNarrow = new Pen(clr);
        Brush brush = new SolidBrush(clr);

        foreach (LineCap lc in Enum.GetValues(typeof(LineCap)))
        {
```
Here's the result:

Keep in mind that I'm using the same enumeration value for the beginning and end of the line. You can use different values if you want.

The NoAnchor value produces the same result as Flat. The SquareAnchor, RoundAnchor, and DiamondAnchor line ends are similar to Square, Round, and Triangle, respectively (as their enumeration values indicate), except that they are larger.

If the various line caps provided by the LineCap enumeration aren't enough for you, you can set the StartCap and/or EndCap properties of the pen equal to LineCap.Custom and then make use of the following properties:

**Pen Properties (selection)**
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomLineCap</td>
<td>CustomStartCap</td>
<td>get/set</td>
</tr>
<tr>
<td>CustomLineCap</td>
<td>CustomEndCap</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `CustomLineCap` class (in `System.Drawing.Drawing2D`) lets you use a path to define the outline of your custom caps. In addition, the `AdjustableArrowCap` class derives from `CustomLineCap` to let you draw arrow caps with more control over the arrow size and filled interior.

Let \( w \) be the width of the line and \( \alpha \) the join angle. It's easy to show that the extension of the miter tip past the actual join point is \( \frac{w}{2}/\sin(\alpha/2) \).
Chapter 18: Edit, List, and Spin

Overview

Just about every Windows program requires a little text input from the user every now and then. Back in Chapter 6, I discussed how your program can install handlers for the KeyDown, KeyUp, and KeyPress events to obtain keyboard input and echo the input back to the user. For many simple purposes, however, you can make use of a type of control traditionally called an edit control, but in the .NET Framework is referred to as a text box.

A text box can range from a small, single-line entry field to a multiline control with word wrap, such as that used in the Microsoft Notepad program. Writing a clone of Notepad is a traditional exercise for a book like this, and we’ll accomplish most of that job in this chapter. (I’ll add printing support in Chapter 21 and drag-and-drop in Chapter 24.)

In this chapter, I’ll also discuss several other important types of controls. The list box presents a scrollable list of options to the user; the combo box combines the text box and the list box. When numbers are involved, you’ll probably want to use the up-down control, more commonly called a spin box.

Single-Line Text Boxes

The simplest text box control is named TextBox, which is derived from the abstract TextBoxBase class, as shown in the class hierarchy on the top of the following page.

Also deriving from TextBoxBase is RichTextBox. The RichTextBox control provides additional functionality over TextBox in the same way that the Microsoft WordPad program provides additional features over Notepad. In TextBox (as in Notepad), you can define a font, but the font applies to the entire document. In RichTextBox (as in WordPad), you can define different fonts as well as other formatting for different parts of the document.

The most important property for text boxes is Text because it contains the text that the user enters into the text box. A program can also initialize the text of the text box and limit the length of the text:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MaxLength</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>TextLength</td>
<td>get</td>
</tr>
<tr>
<td>string[]</td>
<td>Lines</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The TextLength property is the same as Text.Length.
The *Lines* property might also be called *Paragraphs* since it divides the document into text blocks terminated by end-of-line characters. These blocks show up as lines of text if word wrap is off but as paragraphs if word wrap is on.

When you use a text box in a dialog box, in many cases you need to fish the text out of the text box only when the user presses OK. But some dialog boxes like to keep closer track of what the user is entering. Perhaps the dialog box enables the OK button only when the text box contains some valid information. The most useful event for text boxes is actually defined in *Control*:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextChanged</td>
<td>OnTextChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

Here's a program that creates a text box and a label control. By installing a handler for the text box's *TextChanged* event, the program can track the text that the user enters in the text box and replicate it in the label control.

```csharp
class TextBoxDemo : Form
{
    Label label;

    public static void Main()
    {
        Application.Run(new TextBoxDemo());
    }

    public TextBoxDemo()
    {
        Text = "TextBox Demo";

        // Create text box control.
        TextBox txtbox = new TextBox();
textbox.Parent = this;
textbox.Location = new Point(Font.Height, Font.Height);
textbox.Size = new Size(ClientSize.Width - 2 * Font.Height,
                                Font.Height);
textbox.Anchor |= AnchorStyles.Right;
textbox.TextChanged += new EventHandler(TextBoxOnTextChanged);

        // Create label control.
```
Because I've set the Anchor property of the text box, the control will stretch wider when you resize
the form. However, the actual size of the text box doesn't limit the amount of text you can enter. As
you enter text beyond the width of the text box, the text will automatically scroll to the left. When that
happens, the label control may not be able to display all the text even though it has its AutoSize
property set:

```
label = new Label();
label.Parent = this;
label.Location = new Point(Font.Height, 3 * Font.Height);
label.AutoSize = true;
}
void TextBoxOnTextChanged(object obj, EventArgs ea)
{
    TextBox txtbox = (TextBox) obj;

    label.Text = txtbox.Text;
}
```

The default background and foreground colors of the text box are SystemColors.Window and
SystemColors.WindowText, as opposed to the SystemColors.Control and SystemColors.ControlText
colors that the form uses. The text box inherits its default Font property from its parent. The default
BorderStyle property is BorderStyle.Fixed3D, which gives the control a sunken appearance. You can
also use None or FixedSingle.

I want you to take a moment to experiment with the TextBoxDemo program or the Notepad program
or any other text box anywhere in Windows. As you undoubtedly know, the caret in the text box
indicates the insertion point—where text will be inserted when you type. You can move the caret
anywhere you want within the entire block of text by using the cursor arrow keys. You can also
change the location of the caret by clicking with the mouse.

If you hold down the Shift key and move the cursor arrow keys, you define a selection, which is a
block of text that the text box highlights using a reverse-video effect. You can also use the mouse to
define a selection in the text box by dragging the mouse with the button pressed.

Here's the important point: If there's text selected in the text box, the caret is always at the beginning
or the end of the selection. It's important to realize this because the properties that provide
information about the selection also provide information about the caret position. Four properties of TextBoxBase are concerned with the selection and hence also the caret position:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>SelectionStart</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>SelectionLength</td>
<td>get/set</td>
</tr>
<tr>
<td>string</td>
<td>SelectedText</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>HideSelection</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The SelectionStart property is a zero-based index indicating the character position of the beginning of the selection. If SelectionStart is 0, the selection begins at the very beginning of the text in the text box.

The SelectionLength property indicates the number of characters in the selection. If SelectionLength is 0, no text is selected and SelectionStart indicates the caret position. If both SelectionStart and SelectionLength are 0, the caret is located at the very beginning of the text box contents.

If SelectionLength is greater than 0, the precise caret position isn't available. Depending on how you've selected the text (that is, whether you used the left arrow or the right arrow, or you swept the mouse in a particular direction), the caret could be at either the beginning or the end of the selection. The caret position is either SelectionStart or (SelectionStart + SelectionLength).

If there's no selection, SelectedText is the empty string. Otherwise, it's a text string of the selected text. The SelectionLength property is really just a shortcut for SelectedText.Length.

Notice that the SelectedText property is writable. If a program wants to delete the selected text from the text box (without deleting the unselected text), it can simply set the property to an empty string:

textbox.SelectedText = "";

If a program wants to insert text at the caret position, it can call
textbox.SelectedText = "insert text";

If there's currently a selection, the inserted text will replace the selection. If there's no selection, the text will be inserted at the caret position.

The HideSelection property is normally true. That means that when the text box loses the input focus, it no longer highlights the selection. This is normal behavior. Later on in this chapter, I use a nondefault setting of HideSelection in connection with a find-and-replace modeless dialog box.

The TextBox control doesn't have an event that allows a program to determine when the selection changes. However, the RichTextBox control does have an event named SelectionChanged.

The Select and SelectAll methods allow a program to set a selection in a text box:

void Select(int iStart, int iLength)
void SelectAll()
void Clear()

The Select method is equivalent to setting the SelectionStart and SelectionLength properties. The SelectAll method is equivalent to setting SelectionStart to 0 and SelectionLength to TextLength. The Clear method is equivalent to setting Text to an empty string.

By experimenting with TextBoxDemo, you may have noticed that the text box automatically works with the clipboard. You can type Ctrl+X to delete the selected text and copy it to the clipboard, Ctrl+C
to copy selected text to the clipboard without deleting it, and Ctrl+V to paste text from the clipboard.
We'll see how to do this programatically later in the chapter. The standard TextBox also includes
an undo facility. If you select some text and delete it using the Delete key, you can bring it back by
typing Ctrl+Z. It's only a one-level undo: pressing Ctrl+Z again makes the deleted text go away, and
pressing Ctrl+Z again restores the text.

**Multiline Text Boxes**

It's fairly easy to convert a single-line text box to a multiline text box. Basically, you set the Multiline
property to true and (most likely) make the text box large enough to display more than one line.
However, a number of other properties are involved with multiline text boxes, so let's begin by getting
familiar with them.

Here's Multiline and two other related properties:

<table>
<thead>
<tr>
<th>TextBox Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>ScrollBars</td>
</tr>
</tbody>
</table>

The Multiline and WordWrap properties are implemented in TextBoxBase and are also inherited by
RichTextBox. The ScrollBars property is implemented in TextBox and can have one of the following
values:

<table>
<thead>
<tr>
<th>ScrollBars Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Horizontal</td>
</tr>
<tr>
<td>Vertical</td>
</tr>
<tr>
<td>Both</td>
</tr>
</tbody>
</table>

Setting the property to Horizontal, Vertical, or Both makes the scroll bars appear even if they're not
needed, which is the case when there's not enough text to make them useful. However, if the scroll
bars aren't currently needed, they are disabled. Regardless of the setting of the ScrollBars property,
the horizontal scroll bar won't be displayed if WordWrap is set to true.

The RichTextBox control also has a ScrollBars property, but the property is a different type—
RichTextBoxScrollBars—and it lets you get more specific about when you want the scroll bars to
appear:

<table>
<thead>
<tr>
<th>RichTextBoxScrollBars Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Horizontal</td>
</tr>
<tr>
<td>Vertical</td>
</tr>
<tr>
<td>Both</td>
</tr>
<tr>
<td>ForcedHorizontal</td>
</tr>
<tr>
<td>ForcedVertical</td>
</tr>
<tr>
<td>ForcedBoth</td>
</tr>
</tbody>
</table>
The *Horizontal*, *Vertical*, and *Both* members cause the scroll bars to be displayed only when they're needed. The members beginning with the word *Forced* cause the scroll bars to be displayed regardless.

You can't center text vertically in multiline text boxes. The most control you have is with the *TextAlign* property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>HorizontalAlignment</td>
<td>TextAlign</td>
<td>get/set</td>
</tr>
</tbody>
</table>

*HorizontalAlignment* is an enumeration:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>0</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
</tr>
<tr>
<td>Center</td>
<td>2</td>
</tr>
</tbody>
</table>

As you know, the Tab and Enter keys have special meaning in dialog boxes or in any form that has child controls. The Tab key causes the focus to shift among controls; Enter usually activates the OK button. For single-line text boxes implemented in dialog boxes, you probably want the Tab and Enter keys to function normally. For multiline text boxes, however, you probably want the text box itself to capture the Tab and Enter keys. If so, set these two properties to *true*:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>AcceptsReturn</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>AcceptsTab</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>

*AcceptsReturn* is implemented in *TextBox*; *AcceptsTab* is implemented in *TextBoxBase*. If the *AcceptButton* property of the parent form isn't set, the text box will capture the Enter key regardless of the *AcceptsReturn* property.

**Cloning Notepad**

We're now ready to begin building a clone of Notepad. To present the code in manageable chunks, I'm going to build this program up through several levels of inheritance.

Here's the first module, which creates the text box and uses *DockStyle.Fill* to make it fill the form's client area. The constructor concludes by setting several properties appropriate for multiline text boxes.

**NotepadCloneNoMenu.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneNoMenu : Form
{
    protected TextBox textbox;
```
public static void Main()
{
    Application.Run(new NotepadCloneNoMenu());
}

public NotepadCloneNoMenu()
{
    Text = "Notepad Clone No Menu";
    textbox = new TextBox();
    textbox.Parent = this;
    textbox.Dock = DockStyle.Fill;
    textboxBorderStyle = BorderStyle.None;
    textbox.Multiline = true;
    textbox.ScrollBars = ScrollBars.Both;
    textbox.AcceptsTab = true;
}

Considering the length of this program, it has a high level of functionality and works much like the Notepad program when the Word Wrap menu item is checked. Here's what it looks like with some text typed in.

![Screenshot of Notepad Clone No Menu]

Before beginning to implement a menu in Notepad Clone, I want to implement some code that accesses the Windows registry. As you may know, whenever you change the Word Wrap menu item or the font that it uses, the new settings are used when you next run Notepad. You can view all the Notepad registry settings in the Registry Editor (Regedit.exe) under HKEY_CURRENT_USER with the key Software\Microsoft\Notepad.

Besides saving the Word Wrap setting and the font, Notepad also uses the registry to save its window size and position on the desktop. If you move or resize Notepad, terminate the program, and then run it again, it will appear in the saved location and with the saved size.

At first, implementing a feature to save the location and size seems to involve merely saving the form's DesktopBounds property in the registry when the program terminates and setting the property from the registry when the program next runs. Such a scheme would indeed work fine were it not for those window-display options known as minimize and maximize.
Let's examine these cases: When you maximize any Windows program and then restore it, the program returns to the same location and size as before it was maximized. Similarly, when you minimize and then restore the program, it goes back to the same location and size. Windows obviously saves the location and size of the program's window before the window becomes minimized or maximized. A program using the Win32 API has access to this information in the \texttt{WINDOWPLACEMENT} structure (where it is referred to as the \textit{normal position}). The \texttt{WINDOWPLACEMENT} structure is used in the API functions \texttt{GetWindowPlacement} and \texttt{SetWindowPlacement}.

A Windows Forms program doesn't always have direct access to the normal position. When a form is maximized, the \texttt{DesktopBounds} property reflects the \textit{maximized} location and size. If the form has a sizing border, the location of the maximized form has negative coordinates because the form is positioned with the sizing border outside the area of the desktop. When a form is minimized, the \texttt{DesktopBounds} property has special \texttt{X} and \texttt{Y} values of $-32000$, and the \texttt{Height} and \texttt{Width} values represent the size of the minimized button on the Windows taskbar. Only when the form is not minimized or maximized does \texttt{DesktopBounds} reflect the normal position of the form.

Do you see the problem yet? It manifests itself if the program is terminated when it happens to be minimized or maximized. Because \texttt{DesktopBounds} doesn't reflect the normal position of the form, that's not what the program should save in the registry.

For this reason, a Windows Forms program that wants to save its normal position in the registry should maintain a field specifically for that purpose. (In the program coming up shortly, I call this field \texttt{rectNormal}.) The program can set this field from \texttt{DesktopBounds} in its constructor, and it can also reset the field from \texttt{DesktopBounds} during any call to \texttt{OnMove} or \texttt{OnResize} when the form isn't minimized or maximized. Use the \texttt{WindowState} property to determine whether the window is minimized or maximized:

<table>
<thead>
<tr>
<th>Form Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>FormWindowState</td>
</tr>
</tbody>
</table>

The \texttt{FormWindowState} enumeration has the following members:

<table>
<thead>
<tr>
<th>FormWindowState Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>Normal</td>
</tr>
<tr>
<td>Minimized</td>
</tr>
<tr>
<td>Maximized</td>
</tr>
</tbody>
</table>

Here's the next installment in the Notepad Clone series, which derives from NotepadCloneNoMenu and implements the overhead required to access the registry. It also uses the registry to save and restore the location and size of the window. As I mentioned, the \texttt{rectNormal} field is set during the constructor (a time during which the \texttt{WindowState} property is always \texttt{FormWindowState.Normal}) and in the \texttt{OnMove} and \texttt{OnResize} methods whenever \texttt{WindowState} equals \texttt{FormWindowState.Normal}.

\texttt{NotepadCloneWithRegistry.cs}

```csharp
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneWithRegistry: NotepadCloneNoMenu
```
{  
    Rectangle rectNormal;
    protected string strProgName;
    string strRegKey = "Software\ProgrammingWindowsWithCSharp\";
    const string strWinState = "WindowState";
    const string strLocationX = "LocationX";
    const string strLocationY = "LocationY";
    const string strWidth = "Width";
    const string strHeight = "Height";

    public new static void Main()
    {
        Application.Run(new NotepadCloneWithRegistry());
    }

    public NotepadCloneWithRegistry()
    {
        Text = strProgName = "Notepad Clone with Registry";
        rectNormal = DesktopBounds;
    }

    protected override void OnMove(EventArgs ea)
    {
        base.OnMove(ea);

        if (WindowState == FormWindowState.Normal)
            rectNormal = DesktopBounds;
    }

    protected override void OnResize(EventArgs ea)
    {
        base.OnResize(ea);

        if (WindowState == FormWindowState.Normal)
            rectNormal = DesktopBounds;
    }

    protected override void OnLoad(EventArgs ea)
    {
        base.OnLoad(ea);

        // Construct complete registry key.
        strRegKey = strRegKey + strProgName;

        // Load registry information.
        RegistryKey regkey = Registry.CurrentUser.OpenSubKey(strRegKey);
if (regkey != null)
{
    LoadRegistryInfo(regkey);
    regkey.Close();
}
}

protected override void OnClosed(EventArgs ea)
{
    base.OnClosed(ea);

    // Save registry information.

    RegistryKey regkey =
        Registry.CurrentUser.OpenSubKey(strRegKey, true);

    if (regkey == null)
        regkey = Registry.CurrentUser.CreateSubKey(strRegKey);

    SaveRegistryInfo(regkey);
    regkey.Close();
}

protected virtual void SaveRegistryInfo(RegistryKey regkey)
{
    regkey.SetValue(strWinState, (int) WindowState);
    regkey.SetValue(strLocationX, rectNormal.X);
    regkey.SetValue(strLocationY, rectNormal.Y);
    regkey.SetValue(strWidth,       rectNormal.Width);
    regkey.SetValue(strHeight,      rectNormal.Height);
}

protected virtual void LoadRegistryInfo(RegistryKey regkey)
{
    int x  = (int) regkey.GetValue(strLocationX, 100);
    int y  = (int) regkey.GetValue(strLocationY, 100);
    int cx = (int) regkey.GetValue(strWidth, 300);
    int cy = (int) regkey.GetValue(strHeight, 300);

    rectNormal = new Rectangle(x, y, cx, cy);

    // Adjust rectangle for any change in desktop size.

    Rectangle rectDesk = SystemInformation.WorkingArea;

    rectNormal.Width  = Math.Min(rectNormal.Width,  rectDesk.Width);
    rectNormal.Height = Math.Min(rectNormal.Height,  rectDesk.Height);
rectNormal.X -= Math.Max(rectNormal.Right - rectDesk.Right, 0);
rectNormal.Y -= Math.Max(rectNormal.Bottom - rectDesk.Bottom, 0);

// Set form properties.

DesktopBounds = rectNormal;
WindowState = (FormWindowState) regkey.GetValue(strWinState, 0);
}
}

An earlier program in this book that used the registry (DialogsWithRegistry in Chapter 16) loaded information during the form's constructor and saved information during the OnClosed method. The NotepadCloneWithRegistry program, however, loads information during the OnLoad method, which is associated with the Load event and is called after the constructor code executes but before the program is made visible.

I chose this alternative because I wanted each installment in the Notepad Clone series to use its own area of the registry based on its own program name. The constructor in NotepadCloneWithRegistry sets both the strProgName field and the Text property to the string "Notepad Clone with Registry". The next program in the series (coming up soon) is called NotepadCloneWithFile. It sets the strProgName field to "Notepad Clone with File" and initially sets the Text property to "Notepad Clone with File – Untitled".

By the time the OnLoad method in NotepadCloneWithRegistry is called, all the default constructors have been executed. If the NotepadCloneWithRegistry program is running, strProgName will equal "Notepad Clone with Registry". If the NotepadCloneWithRegistry program is running, strProgName will equal "Notepad Clone with File". The OnLoad method uses strProgName to form a registry key that is also later used in the OnClosed method.

The OnLoad method in NotepadCloneWithRegistry calls the virtual method LoadRegistryInfo, and the OnClosed method calls another virtual method named SaveRegistryInfo. Both these virtual methods have RegistryKey arguments. Thus, any subsequent program in the Notepad Clone series can simply override these two virtual methods to load and save registry information. The overrides must also call the base methods. We'll see how this works shortly.

SaveRegistryInfo saves the WindowState property and the four components of the rectNormal field. LoadRegistryInfo first sets the rectNormal field from these four components. However, it could be that the user changed the desktop size after closing the earlier instance of the program. If so, rectNormal is adjusted to fit within the desktop. The method sets the DesktopBounds property from the adjusted rectNormal field and then sets the WindowState property from the value saved in the registry. Thus, if you close the program in a minimized or maximized state, that state will be reflected the next time the program is run. However, if you then restore the program to its normal size, the location and size will be the same as the previous instance, before it was minimized or maximized. (This behavior is a little different from the Windows Notepad program. Notepad always comes up in its normal state regardless of how the previous instance was closed.)

**The Notepad Clone with File I/O**

The next step for the Notepad Clone series is implementing a File menu in the program. We've seen file I/O code before, but because I'm trying to write a "real-life" program here, Notepad Clone must do what other programs dealing with documents do, which is sometimes display a message box saying, "The text in the file has changed. Do you want to save the changes?"

The following property of TextBoxBase is useful for implementing such a facility:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>bool</td>
<td>get/set</td>
</tr>
</tbody>
</table>
When you create a new TextBox or RichTextBox, the Modified property is initialized to false. Thereafter, whenever the user does something to change the contents of the text box (such as typing), the text box sets its Modified property to true. Your program uses this property to determine whether the contents of the text box must be saved to a file. Whenever your program saves the contents of the text box to a file, it must reset the property to false. (An indicator that the text has changed is sometimes called a dirty bit.)

On several occasions, a program dealing with documents should display a message box asking whether the document should be saved: when the user selects New from the File menu, essentially deleting the existing file from the application; when the user selects Open from the File menu, replacing the existing file; and when the user wants to exit the program. In the NotepadCloneWithFile program (coming up soon), the method that checks whether it's OK to delete the existing document is called OkToTrash.

If the user chooses to save the existing document to a file, the program normally must perform the same operation as if the user had selected Save from the File menu. However, if the document has no name, the Save As dialog box must be displayed. If the user then clicks the Cancel button in the dialog box, the file isn't saved, but neither does the program perform the New, Open, or Exit operation.

To handle the last case, the program must override the OnClosing method, which occurs before the form is closed, after which the OnClosed method is called. The OnClosing method is accompanied by a CancelEventArgs object, which has a single Boolean property (named Cancel) that the OnClosing method can set to true to prevent the program from being closed.

NotepadCloneWithFile also requires a method of TextBoxBase that I haven't mentioned yet:

**TextBoxBase Methods (selection)**

```csharp
void ClearUndo()
```

Some background is required here. As I mentioned, the TextBoxBase class implements a single-level undo feature. When a change is made to the contents of the text box, the previous version is often saved. As we'll see in the next section, TextBoxBase also includes a Boolean property named CanUndo that returns true to indicate the existence of a previous version and a method named Undo that reverts to the previous version.

However, at times you don't want the user to be able to revert to the previous contents of the text box, for example, when the program loads the text box from a file. It doesn't make much sense for the Undo command to revert to the text box state before the file was loaded. In such cases, the program calls ClearUndo to delete the previous version of the text box contents.

Here's the version of Notepad Clone that implements a File menu. Currently, three options involving printing are left unimplemented. I'll be supplying code for those items in Chapter 21, in the NotepadCloneWithPrinting program.

**NotepadCloneWithFile.cs**

```csharp
// NotepadCloneWithFile.cs © 2001 by Charles Petzold
using Microsoft.Win32;        // For registry classes
using System;
using System.ComponentModel;  // For CancelEventArgs class
using System.Drawing;
using System.IO;
using System.Text;            // For Encoding class
using System.Windows.Forms;
```
class NotepadCloneWithFile: NotepadCloneWithRegistry
{
    // Fields
    protected string strFileName;
    const string strEncoding = "Encoding"; // For registry
    const string strFilter = "Text Documents(*.txt)|*.txt|All Files(*.*)|*.*";
    MenuItem miEncoding;
    MenuItemEncoding mieChecked;

    // Entry point
    public new static void Main()
    {
        Application.Run(new NotepadCloneWithFile());
    }

    // Constructor
    public NotepadCloneWithFile()
    {
        strProgName = "Notepad Clone with File";
        MakeCaption();
        Menu = new MainMenu();

        // File menu
        MenuItem mi = new MenuItem("&File");
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

        // File New
        mi = new MenuItem("&New");
        mi.Click += new EventHandler(MenuFileNewOnClick);
        mi.Shortcut = Shortcut.CtrlN;
        Menu.MenuItems[index].MenuItems.Add(mi);

        // File Open
        MenuItem miFileOpen = new MenuItem("&Open...");
        miFileOpen.Click += new EventHandler(MenuFileOpenOnClick);
        miFileOpenShortcut = Shortcut.CtrlO;
        Menu.MenuItems[index].MenuItems.Add(miFileOpen);

        // File Save
        MenuItem miFileSave = new MenuItem("&Save");
    }
}
miFileSave.Click += new EventHandler(MenuFileSaveOnClick);
miFileSave.Shortcut = Shortcut.CtrlS;
Menu.MenuItems[index].MenuItems.Add(miFileSave);

    // File Save As

mi = new MenuItem("Save &As...");
mi.Click += new EventHandler(MenuFileSaveAsOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

    // File Encoding

miEncoding = new MenuItem("&Encoding");
Menu.MenuItems[index].MenuItems.Add(miEncoding);
Menu.MenuItems[index].MenuItems.Add("-");

    // File Encoding submenu

EventHandler eh = new EventHandler(MenuFileEncodingOnClick);
string[] astrEncodings = { "&ASCII", "&Unicode",
                        "&Big-Endian Unicode",
                        "UTF-&7", "&UTF-&8" };
Encoding[] aenc = { Encoding.ASCII, Encoding.Unicode,
                  Encoding.BigEndianUnicode,
                  Encoding.UTF7, Encoding.UTF8 };

for (int i = 0; i < astrEncodings.Length; i++)
{
    MenuItemEncoding mie = new MenuItemEncoding();
    mie.Text = astrEncodings[i];
    mie.Encoding = aenc[i];
    mie.RadioCheck = true;
    mie.Click += eh;

    miEncoding.MenuItems.Add(mie);
}
mieChecked = (MenuItemEncoding) miEncoding.MenuItems[4];  // UTF-8
mieChecked.Checked = true;

    // File Page Setup

mi = new MenuItem("Page Set&up...");
mi.Click += new EventHandler(MenuFileSetupOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

// File Print Preview

mi = new MenuItem("Print Pre&view...");
mi.Click += new EventHandler(MenuFilePreviewOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

// File Print

mi = new MenuItem("&Print...");
mi.Click += new EventHandler(MenuFilePrintOnClick);
mi.Shortcut = Shortcut.CtrlP;
Menu.MenuItems[index].MenuItems.Add(mi);
Menu.MenuItems[index].MenuItems.Add("-");

// File Exit

mi = new MenuItem("E&xit");
mi.Click += new EventHandler(MenuFileExitOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

// Set system event.

SystemEvents.SessionEnding +=
new SessionEndingEventHandler(OnSessionEnding);

protected override void OnLoad(EventArgs ea)
{
    base.OnLoad(ea);

    // Deal with the command-line argument.

    string[] astrArgs = Environment.GetCommandLineArgs();

    if (astrArgs.Length > 1) // First argument is program name!
    {
        if (File.Exists(astrArgs[1]))
        {
            LoadFile(astrArgs[1]);
        } else
        {
            DialogResult dr =
MessageBox.Show("Cannot find the " +
    Path.GetFileName(astrArgs[1]) +
    " file.
    " +
    "Do you want to create a new file?",
    strProgName,
    MessageBoxButtons.YesNoCancel,
    MessageBoxIcon.Question);

switch(dr)
{
        File.Create(strFileName = astrArgs[1]).Close();
        MakeCaption();
        break;

    case DialogResult.No:
        break;

    case DialogResult.Cancel:
        Close();
        break;
}

protected override void OnClosing(CancelEventArgs cea)
{
    base.OnClosing(cea);

    cea.Cancel = !OkToTrash();
}

// Event handlers
void OnSessionEnding(object obj, SessionEndingEventArgs seea)
{
    seea.Cancel = !OkToTrash();
}

// Menu items
void MenuFileNewOnClick(object obj, EventArgs ea)
{
    if (!OkToTrash())
        return;

    textbox.Clear();
    textbox.ClearUndo();
    textbox.Modified = false;
strFileName = null;
MakeCaption();
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
    if (!OkToTrash())
        return;

    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Filter = strFilter;
    ofd.FileName = "*.txt";

    if (ofd.ShowDialog() == DialogResult.OK)
        LoadFile(ofd.FileName);
}
void MenuFileEncodingOnClick(object obj, EventArgs ea)
{
    mieChecked.Checked = false;
    mieChecked = (MenuItemEncoding) obj;
    mieChecked.Checked = true;
}
void MenuFileSaveOnClick(object obj, EventArgs ea)
{
    if (strFileName == null || strFileName.Length == 0)
        SaveFileDlg();
    else
        SaveFile();
}
void MenuFileSaveAsOnClick(object obj, EventArgs ea)
{
    SaveFileDlg();
}
protected virtual void MenuFileSetupOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Page Setup not yet implemented!", strProgName);
}
protected virtual void MenuFilePreviewOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Print Preview not yet implemented!", strProgName);
}
protected virtual void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Print not yet implemented!", strProgName);
void MenuFileExitOnClick(object obj, EventArgs ea)
{
    if (OkToTrash())
        Application.Exit();
}

protected override void LoadRegistryInfo(RegistryKey regkey)
{
    base.LoadRegistryInfo(regkey);

    // Set encoding setting.

    int index = (int) regkey.GetValue(strEncoding, 4);

    mieChecked.Checked = false;
    mieChecked = (MenuItemEncoding) miEncoding.MenuItems[index];
    mieChecked.Checked = true;
}

protected override void SaveRegistryInfo(RegistryKey regkey)
{
    base.SaveRegistryInfo(regkey);
    regkey.SetValue(strEncoding, mieChecked.Index);
}

// Utility routines

protected void LoadFile(string strFileName)
{
    StreamReader sr;

    try
    {
        sr = new StreamReader(strFileName);
    }
    catch (Exception exc)
    {
        MessageBox.Show(exc.Message, strProgName,
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Asterisk);

        return;
    }
    textbox.Text = sr.ReadToEnd();
    sr.Close();

    this.strFileName = strFileName;
void SaveFile()
{
    try
    {
        StreamWriter sw = new StreamWriter(strFileName, false, mieChecked.Encoding);
        sw.Write(txtbox.Text);
        sw.Close();
    }
    catch (Exception exc)
    {
        MessageBox.Show(exc.Message, strProgName,
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Asterisk);
        return;
    }
    textbox.Modified = false;
}

bool SaveFileDialog()
{
    SaveFileDialog sfd = new SaveFileDialog();

    if (strFileName != null && strFileName.Length > 1)
        sfd.FileName = strFileName;
    else
        sfd.FileName = "*.txt";

    sfd.Filter = strFilter;

    if (sfd.ShowDialog() == DialogResult.OK)
    {
        strFileName = sfd.FileName;
        SaveFile();
        MakeCaption();
        return true;
    }
    else
    {

return false;       // Return values are for OkToTrash.

protected void MakeCaption()
{
    Text = strProgName + " - " + FileTitle();
}

protected string FileTitle()
{
    return (strFileName != null && strFileName.Length > 1) ?
            Path.GetFileName(strFileName) : "Untitled";
}

protected bool OkToTrash()
{
    if (!txtbox.Modified)
        return true;

    DialogResult dr =
        MessageBox.Show("The text in the " + FileTitle() + " file has changed.\r\n\r\n" + "Do you want to save the changes?", strProgName,
                    MessageBoxButtons.YesNoCancel,
                    MessageBoxIcon.Exclamation);

    switch (dr)
    {
    case DialogResult.Yes:
        return SaveFileDlg();

    case DialogResult.No:
        return true;

    case DialogResult.Cancel:
        return false;
    }
    return false;
}

class MenuItemEncoding: MenuItem
{
    public Encoding Encoding;
}

Although using the File Open menu option is the most common way of loading a file into Notepad, you can also run Notepad from a command line and specify a file as an argument. NotepadCloneWithFile likewise overrides the OnLoad method and attempts to load a file specified as a command-line argument.
The Notepad program includes a special combo box in the Save As dialog box that lets you specify a file encoding. See it at the bottom? The options are ANSI, Unicode, Unicode Big Endian, and UTF-8. (If you're unfamiliar with these terms and the encoding of text files under Windows, you'll probably want to read the section on "Reading and Writing Text" in Appendix A.)

Unfortunately, Windows Forms programs don't have quite the same flexibility as Windows API programs in enhancing the common dialog boxes. For this reason, I've added an Encoding item to the File menu. This item invokes another submenu that lists five encoding options: ASCII, Unicode, Big-Endian Unicode, UTF-7, and UTF-8. A small MenuItemEncoding class overrides MenuItem so that the appropriate object of the Encoding class can be stored along with each of these five items.

NotepadCloneWithFile overrides the SaveRegistryInfo and LoadRegistryInfo methods in NotepadCloneWithRegistry to save and later reload the character encoding the user selects. Notice that these methods call the methods in the base class so that the program continues to save and load the location and size of the window.

**Notepad Clone Continued**

The next program we're going to look at is NotepadCloneWithEdit. This program derives from NotepadCloneWithFile and implements an Edit menu. The Edit menu would be more difficult than it is (and would require material I won't cover until Chapter 24) were it not for the built-in support that TextBoxBase has for the clipboard. Even with the earliest program in the Notepad Clone series, you can type Ctrl+Z, Ctrl+X, Ctrl+C, and Ctrl+V to perform undo, cut, copy, and paste operations.

As I mentioned earlier, the undo operation restores the text of the text box to a previous version. Both the cut and copy operations copy the selected text to the clipboard. In addition, the cut operation deletes the text. The paste operation copies text from the clipboard to the text box. If text is selected at the time of the paste operation, the selection is replaced by the pasted text.

Although TextBoxBase supports these operations directly, Undo, Cut, Copy, and Paste should also be options on the Edit menu. In addition, a Delete item (with a Delete key shortcut) should delete selected text without copying it to the clipboard. (In some applications, a Clear menu item does the work of Delete.) Interestingly enough, if you simply create an Edit menu with these items—and you specify the appropriate Shortcut properties—you effectively disable the ability of the text box to respond to these keystrokes! The reason for this behavior is that the menu gets priority over keystrokes, and if (for example) Ctrl+C is a shortcut in a menu item, that keystroke is consumed by the menu and never reaches the text box.

You can get the shortcuts to work in the text box again by installing Click event handlers for the Undo, Cut, Copy, Paste, and Delete menu items and by calling the following five methods in response to the events:

**TextBoxBase Methods (selection)**

```csharp
void Undo()
void Cut()
```
void Copy()
void Paste()
void Clear()

You'll also want to install a Popup event handler for the Edit menu to enable and disable these five items appropriately. You can enable the Undo item based on the return value of the following property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>CanUndo</td>
<td>get</td>
</tr>
</tbody>
</table>

Enable the Cut, Copy, and Delete (or Clear) menu items only if the SelectionLength property is greater than 0. Otherwise, there's no selected text to delete or copy to the clipboard.

The tough one is Paste. The Paste menu item should be enabled only if there's text in the clipboard. Until Chapter 24, I'm afraid you'll just have to take my word for it that you should enable the Paste item if the following expression returns true:

```csharp
Clipboard.GetDataObject().GetDataPresent(typeof(string))
```

GetDataObject is a static method of Clipboard that returns an object of type IDataObject. GetDataPresent is a method of that interface that returns true if the clipboard contains data of the type specified by its argument, which in this case is a string.

If the Edit menu contained only Undo, Cut, Copy, Paste, and Delete, it would be a snap. Even a Select All option can be implemented by a call to SelectAll, and the Time/Date item found on Notepad's Edit menu is also easy using the ToString method of the DateTime class.

But Notepad's Edit menu also includes Find, Find Next, Replace, and Go To, the latter of which lets the user jump to a particular line of the document. I decided not to implement the Go To item, primarily because the text box controls are missing a method that indicates the character offset of a specified line. (In Win32 API programming, this facility is provided by the EM_LINEINDEX message.) I'll make up for this omission in the Format menu, when I allow you to do something Notepad doesn't: select a text color and a background color.

The following file supports dialog boxes for Find and Replace. Both FindDialog and ReplaceDialog are descended from the abstract class FindReplaceDialog. The abstract class creates all the controls common to both dialog boxes. The two classes descended from FindReplaceDialog disable certain of these controls and (in one case) move a button to a more appropriate location.

FindReplaceDialog.cs

```csharp
//------------------------------------------------
// FindReplaceDialog.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FindDialog: FindReplaceDialog
{
    public FindDialog()
    {
        Text = "Find";
    }
```
labelReplace.Visible = false;
txtboxReplace.Visible = false;
btnReplace.Visible = false;
btnReplaceAll.Visible = false;
btnCancel.Location = btnReplace.Location;
}
}
class ReplaceDialog: FindReplaceDialog
{
    public ReplaceDialog()
    {
        Text = "Replace";
        grpboxDirection.Visible = false;
    }
}
abstract class FindReplaceDialog: Form
{

    // Fields
    protected Label labelFind, labelReplace;
    protected TextBox txtboxFind, txtboxReplace;
    protected CheckBox chkboxMatchCase;
    protected GroupBox grpboxDirection;
    protected RadioButton radiobtnUp, radiobtnDown;
    protected Button btnFindNext, btnReplace, btnReplaceAll, btnCancel;

    // Public events
    public event EventHandler FindNext;
    public event EventHandler Replace;
    public event EventHandler ReplaceAll;
    public event EventHandler CloseDlg;

    public FindReplaceDialog()
    {
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox = false;
        MinimizeBox = false;
        MaximizeBox = false;
        ShowInTaskbar = false;
        StartPosition = FormStartPosition.Manual;
        Location = ActiveForm.Location + SystemInformation.CaptionButtonSize + SystemInformation.FrameBorderSize;

        labelFind = new Label();
    }
labelFind.Parent = this;
labelFind.Text = "Find what:";
labelFind.Location = new Point(8, 8);
labelFind.Size = new Size(64, 8);

txtboxFind = new TextBox();
textboxFind.Parent = this;
textboxFind.Location = new Point(72, 8);
textboxFind.Size = new Size(136, 8);
textboxFind.TextChanged +=
    new EventHandler(TextboxFindOnTextChanged);

labelReplace = new Label();
labelReplace.Parent = this;
labelReplace.Text = "Replace with:";
labelReplace.Location = new Point(8, 24);
labelReplace.Size = new Size(64, 8);

txtboxReplace = new TextBox();
textboxReplace.Parent = this;
textboxReplace.Location = new Point(72, 24);
textboxReplace.Size = new Size(136, 8);

chkboxMatchCase = new CheckBox();
chkboxMatchCase.Parent = this;
chkboxMatchCase.Text = "Match case";
chkboxMatchCase.Location = new Point(8, 50); // 48);
chkboxMatchCase.Size = new Size(64, 8);

grpboxDirection = new GroupBox();
grpboxDirection.Parent = this;
grpboxDirection.Text = "Direction";
grpboxDirection.Location = new Point(100, 40);
grpboxDirection.Size = new Size(96, 24);

radiobtnUp = new RadioButton();
radiobtnUp.Parent = grpboxDirection;
radiobtnUp.Text = "&Up";
radiobtnUp.Location = new Point(8, 8);
radiobtnUp.Size = new Size(32, 12);

radiobtnDown = new RadioButton();
radiobtnDown.Parent = grpboxDirection;
radiobtnDown.Text = "&Down";
radiobtnDown.Location = new Point(40, 8);
radiobtnDown.Size = new Size(40, 12);

btnFindNext = new Button();
btnFindNext.Parent = this;
btnFindNext.Text = "&Find Next";
btnFindNext.Enabled = false;
btnFindNext.Location = new Point(216, 4);
btnFindNext.Size = new Size(64, 16);
btnFindNext.Click += new EventHandler(ButtonFindNextOnClick);

btnReplace = new Button();
btnReplace.Parent = this;
btnReplace.Text = "&Replace";
btnReplace.Enabled = false;
btnReplace.Location = new Point(216, 24);
btnReplace.Size = new Size(64, 16);
btnReplace.Click += new EventHandler(ButtonReplaceOnClick);

btnReplaceAll = new Button();
btnReplaceAll.Parent = this;
btnReplaceAll.Text = "Replace &All";
btnReplaceAll.Enabled = false;
btnReplaceAll.Location = new Point(216, 44);
btnReplaceAll.Size = new Size(64, 16);
btnReplaceAll.Click += new EventHandler(ButtonReplaceAllOnClick);

btnCancel = new Button();
btnCancel.Parent = this;
btnCancel.Text = "Cancel";
btnCancel.Location = new Point(216, 64);
btnCancel.Size = new Size(64, 16);
btnCancel.Click += new EventHandler(ButtonCancelOnClick);
CancelButton = btnCancel;

ClientRect = new Size(288, 84);
AutoScaleBaseSize = new Size(4, 8);
}

// Properties

public string FindText
{
    set { txtboxFind.Text = value; }
    get { return txtboxFind.Text; }
}

public string ReplaceText
{
```csharp
set { textboxReplace.Text = value; }
get { return textboxReplace.Text; }
}

public bool MatchCase
{
    set { chkboxMatchCase.Checked = value; }
get { return chkboxMatchCase.Checked; }
}

public bool FindDown
{
    set
    {
        if (value)
        {
            radiobtnUp.Checked = true;
        }
        else
        {
            radiobtnDown.Checked = true;
        }
    }
get { return radiobtnDown.Checked; }
}

    // Event handlers

void TextBoxFindOnTextChanged(object obj, EventArgs ea)
{
    btnFindNext.Enabled =
    btnReplace.Enabled =
    btnReplaceAll.Enabled = txtboxFind.Text.Length > 0;
}

void ButtonFindNextOnButtonClick(object obj, EventArgs ea)
{
    if (FindNext != null)
    {
        FindNext(this, EventArgs.Empty);
    }
}

void ButtonReplaceOnButtonClick(object obj, EventArgs ea)
{
    if (Replace != null)
    {
        Replace(this, EventArgs.Empty);
    }
}

void ButtonReplaceAllOnButtonClick(object obj, EventArgs ea)
{
    if (ReplaceAll != null)
    {
        ReplaceAll(this, EventArgs.Empty);
    }
}

void ButtonCancelOnButtonClick(object obj, EventArgs ea)
{
    if (CloseDlg != null)
    {
        CloseDlg(this, EventArgs.Empty);
    }
}
```
FindReplaceDialog has four properties that give the main program access to the text of two text boxes (the "find" text and the "replace" text), the Match Case check box, and the pair of radio buttons for specifying a search up or down in the document.

FindDialog and ReplaceDialog are both intended to be modeless dialog boxes. Indeed, these are classic examples of dialog boxes that work best by being modeless. You want the dialog box to be up and visible as you perform a repetitive search or replace operation. As you know from Chapter 16, modeless dialog boxes usually communicate to the underlying program through events. FindReplaceDialog defines four public events:

```csharp
public event EventHandler FindNext;
public event EventHandler Replace;
public event EventHandler ReplaceAll;
public event EventHandler CloseDlg;
```

These events are triggered when the user presses the dialog box buttons labeled Find Next, Replace, Replace All, and Cancel.

Here's NotepadCloneWithEdit, which adds an Edit menu and implements all the items on that menu.

```csharp
class NotepadCloneWithEdit: NotepadCloneWithFile
{
    MenuItem miEditUndo, miEditCut, miEditCopy, miEditPaste, miEditDelete;
    string   strFind = "", strReplace = "";
    bool     bMatchCase = false, bFindDown = true;

    public new static void Main()
    {
        Application.Run(new NotepadCloneWithEdit());
    }
    public NotepadCloneWithEdit()
    {
        strProgName = "Notepad Clone with Edit";
        MakeCaption();

        // Edit menu

        MenuItem mi = new MenuItem("&Edit");
```

```csharp
```
mi.Popup += new EventHandler(MenuEditOnPopup);
Menu.MenuItems.Add(mi);
int index = Menu.MenuItems.Count - 1;

    // Edit Undo menu item

miEditUndo = new MenuItem("&Undo");
miEditUndo.Click += new EventHandler(MenuEditUndoOnClick);
miEditUndo.Shortcut = Shortcut.CtrlZ;
Menu.MenuItems[index].MenuItems.Add(miEditUndo);
Menu.MenuItems[index].MenuItems.Add("-");

    // Edit Cut menu item

miEditCut = new MenuItem("Cu&t");
miEditCut.Click += new EventHandler(MenuEditCutOnClick);
miEditCut.Shortcut = Shortcut.CtrlX;
Menu.MenuItems[index].MenuItems.Add(miEditCut);

    // Edit Copy menu item

miEditCopy = new MenuItem("&Copy");
miEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
miEditCopy.Shortcut = Shortcut.CtrlC;
Menu.MenuItems[index].MenuItems.Add(miEditCopy);

    // Edit Paste menu item

miEditPaste = new MenuItem("&Paste");
miEditPaste.Click += new EventHandler(MenuEditPasteOnClick);
miEditPaste.Shortcut = Shortcut.CtrlV;
Menu.MenuItems[index].MenuItems.Add(miEditPaste);

    // Edit Delete menu item

miEditDelete = new MenuItem("De&lete");
miEditDelete.Click += new EventHandler(MenuEditDeleteOnClick);
miEditDelete.Shortcut = Shortcut.Del;
Menu.MenuItems[index].MenuItems.Add(miEditDelete);
Menu.MenuItems[index].MenuItems.Add("-");

    // Edit Find menu item

mi = new MenuItem("&Find...");
mi.Click += new EventHandler(MenuEditFindOnClick);
mi.Shortcut = Shortcut.CtrlF;
Menu.MenuItems[index].MenuItems.Add(mi);

    // Edit Find Next menu item

mi = new MenuItem("Find &Next");
mi.Click += new EventHandler(MenuEditFindNextOnClick);
mi.Shortcut = Shortcut.F3;
Menu.MenuItems[index].MenuItems.Add(mi);

    // Edit Replace menu item

mi = new MenuItem("&Replace...");
mi.Click += new EventHandler(MenuEditReplaceOnClick);
mi.Shortcut = Shortcut.CtrlH;
Menu.MenuItems[index].MenuItems.Add(mi);
Menu.MenuItems[index].MenuItems.Add("-");

    // Edit Select All menu item

mi = new MenuItem("Select &All");
mi.Click += new EventHandler(MenuEditSelectAllOnClick);
mi.Shortcut = Shortcut.CtrlA;
Menu.MenuItems[index].MenuItems.Add(mi);

    // Edit Time/Date menu item

mi = new MenuItem("Time/&Date");
mi.Click += new EventHandler(MenuEditTimeDateOnClick);
mi.Shortcut = Shortcut.F5;
Menu.MenuItems[index].MenuItems.Add(mi);
}
void MenuEditOnPopup(object obj, EventArgs ea)
{
    miEditUndo.Enabled = textbox.CanUndo;

    miEditCut.Enabled = miEditCopy.Enabled = miEditDelete.Enabled = (textbox.SelectionLength > 0);

    miEditPaste.Enabled = Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
void MenuEditUndoOnClick(object obj, EventArgs ea)
{

void MenuEditCutOnClick(object obj, EventArgs ea)
{
    textbox.Cut();
}

void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    textbox.Copy();
}

void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    textbox.Paste();
}

void MenuEditDeleteOnClick(object obj, EventArgs ea)
{
    textbox.Clear();
}

void MenuEditFindOnClick(object obj, EventArgs ea)
{
    if (OwnedForms.Length > 0)
        return;

    textbox.HideSelection = false;

    FindDialog dlg = new FindDialog();

    dlg.Owner = this;
    dlg.FindText = strFind;
    dlg.MatchCase = bMatchCase;
    dlg.FindDown = bFindDown;
    dlg.FindNext += new EventHandler(FindDialogOnFindNext);
    dlg.CloseDlg += new EventHandler(FindReplaceDialogOnCloseDlg);
    dlg.Show();
}

void MenuEditFindNextOnClick(object obj, EventArgs ea)
{
    if (strFind.Length == 0)
    {
        if (OwnedForms.Length > 0)
            return;

        MenuEditFindOnClick(obj, ea);
    }
}
else
    FindNext();
}

void MenuEditReplaceOnClick(object obj, EventArgs ea)
{
    if (OwnedForms.Length > 0)
        return;

    textbox.HideSelection = false;

    ReplaceDialog dlg = new ReplaceDialog();
    dlg.Owner = this;
    dlg.FindText = strFind;
    dlg.ReplaceText = strReplace;
    dlg.MatchCase = bMatchCase;
    dlg.FindDown = bFindDown;
    dlg.FindNext += new EventHandler(FindDialogOnFindNext);
    dlg.Replace += new EventHandler(ReplaceDialogOnReplace);
    dlg.ReplaceAll += new EventHandler(ReplaceDialogOnReplaceAll);
    dlg.CloseDlg += new EventHandler(FindReplaceDialogOnCloseDlg);
    dlg.Show();
}

void MenuEditSelectAllOnClick(object obj, EventArgs ea)
{
    textbox.SelectAll();
}

void MenuEditTimeDateOnClick(object obj, EventArgs ea)
{
    DateTime dt = DateTime.Now;
    textbox.SelectedText = dt.ToString("t") + " " + dt.ToString("d");
}

void FindDialogOnFindNext(object obj, EventArgs ea)
{
    FindReplaceDialog dlg = (FindReplaceDialog) obj;
    strFind = dlg.FindText;
    bMatchCase = dlg.MatchCase;
    bFindDown = dlg.FindDown;

    FindNext();
}

bool FindNext()
{
    if (bFindDown)
int iStart = txtbox.SelectionStart + 
  txtbox.SelectionLength;

while (iStart + strFind.Length <= txtbox.TextLength)
{
    if (string.Compare(strFind, 0, txtbox.Text, iStart,
            strFind.Length, !bMatchCase) == 0)
    {
        txtbox.SelectionStart = iStart;
        txtbox.SelectionLength = strFind.Length;
        return true;
    }
    iStart++;
}

else
{
    int iStart = txtbox.SelectionStart - strFind.Length;

    while (iStart >= 0)
    {
        if (string.Compare(strFind, 0, txtbox.Text, iStart,
                strFind.Length, !bMatchCase) == 0)
        {
            txtbox.SelectionStart = iStart;
            txtbox.SelectionLength = strFind.Length;
            return true;
        }
        iStart--;
    }
}

MessageBox.Show("Cannot find \\
\"\" + strFind + \\
\"\", strProgName,
        MessageBoxButtons.OK,
        MessageBoxIcon.Exclamation);
    return false;
}

void ReplaceDialogOnReplace(object obj, EventArgs ea)
{
    FindReplaceDialog dlg = (FindReplaceDialog) obj;

    strFind    = dlg.FindText;
    strReplace = dlg.ReplaceText;
    bMatchCase = dlg.MatchCase;
if (string.Compare(strFind, textbox.SelectedText, !bMatchCase) == 0)
{
    textbox.SelectedText = strReplace;
}
FindNext();
}
void ReplaceDialogOnReplaceAll(object obj, EventArgs ea)
{
    FindReplaceDialog dlg = (FindReplaceDialog) obj;
    string str = textbox.Text;
    strFind = dlg.FindText;
    strReplace = dlg.ReplaceText;
    bMatchCase = dlg.MatchCase;
    
    if (bMatchCase)
    {
        str = str.Replace(strFind, strReplace);
    }
    else
    {
        for (int i = 0; i < str.Length - strFind.Length; )
        {
            if (String.Compare(str, i, strFind, 0,
                strFind.Length, true) == 0)
            {
                str = str.Remove(i, strFind.Length);
                str = str.Insert(i, strReplace);
                i += strReplace.Length;
            }
            else
            {
                i += 1;
            }
        }
    }
    if (str != textbox.Text)
    {
        textbox.Text = str;
        textbox.SelectionStart = 0;
        textbox.SelectionLength = 0;
        textbox.Modified = true;
    }
}
void FindReplaceDialogOnCloseDlg(object obj, EventArgs ea)
When a menu item invokes a modeless dialog box, you usually don't want the item to invoke multiple copies of the dialog box. For that reason, the Find and Replace menu items check whether the `OwnedForms` property (which returns an array of `Form` objects) has a `Length` property greater than 0. If it does, one of the modeless dialog boxes is up and another shouldn't be displayed. As usual, the `Owner` property of the modeless dialog box is assigned `this`, referring to the program itself.

When the program creates either of the two modeless dialog boxes, it also installs event handlers for the events implemented by the dialog boxes. It's in response to these events that the program does most of its work. For the actual search logic, the program uses the `Compare` method of the `String` class. For replacing text, it uses `Remove` and `Insert`. These methods are described in Appendix C.

The next installment in the Notepad Clone series is comparatively simple. It implements the Format menu, which expands on the Notepad functionality by including Background Color along with the standard Word Wrap and Font options. Word Wrap is simply a checked menu item, while the Font and Background Color items make use of `FontDialog` and `ColorDialog`.

**NotepadCloneWithFormat.cs**

```
//-----------------------------------------------------
// NotepadCloneWithFormat.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadCloneWithFormat: NotepadCloneWithEdit
{
    // Strings for registry
    const string strWordWrap = "WordWrap";
    const string strFontFace = "FontFace";
    const string strFontSize = "FontSize";
    const string strFontStyle = "FontStyle";
    const string strForeColor = "ForeColor";
    const string strBackColor = "BackColor";
    const string strCustomClr = "CustomColor";

    ColorDialog clrdlg = new ColorDialog();
    MenuItem   miFormatWrap;

    public new static void Main()
{
        Application.Run(new NotepadCloneWithFormat());
    }
}
```
strProgName = "Notepad Clone with Format";
MakeCaption();

// Format

MenuItem mi = new MenuItem("Format");
mi.Popup += new EventHandler(MenuFormatOnPopup);
Menu.MenuItems.Add(mi);
int index = Menu.MenuItems.Count - 1;

// Format Word Wrap

miFormatWrap = new MenuItem("&Word Wrap");
miFormatWrap.Click += new EventHandler(MenuFormatWrapOnClick);
Menu.MenuItems[index].MenuItems.Add(miFormatWrap);

// Format Font

mi = new MenuItem("&Font...");
mi.Click += new EventHandler(MenuFormatFontOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);

// Format Background Color

mi = new MenuItem("Background &Color...");
mi.Click += new EventHandler(MenuFormatColorOnClick);
Menu.MenuItems[index].MenuItems.Add(mi);
}
protected override void OnLoad(EventArgs ea)
{
    base.OnLoad(ea);

    // Help

    MenuItem mi = new MenuItem("&Help");
    Menu.MenuItems.Add(mi);
    int index = Menu.MenuItems.Count - 1;

    // Help About

    mi = new MenuItem("&About " + strProgName + "...");
    mi.Click += new EventHandler(MenuHelpAboutOnClick);
    Menu.MenuItems[index].MenuItems.Add(mi);
}
void MenuFormatOnPopup(object obj, EventArgs ea)
{ miFormatWrap.Checked = textbox.WordWrap;
}

void MenuFormatWrapOnClick(object obj, EventArgs ea)
{
    MenuItem mi = (MenuItem) obj;
    mi.Checked ^= true;
    textbox.WordWrap = mi.Checked;
}

void MenuFormatFontOnClick(object obj, EventArgs ea)
{
    FontDialog fontdlg = new FontDialog();
    fontdlg.ShowDialog();
    fontdlg.Font = textbox.Font;
    fontdlg.Color = textbox.ForeColor;
    if (fontdlg.ShowDialog() == DialogResult.OK)
    {
        textbox.Font = fontdlg.Font;
        textbox.ForeColor = fontdlg.Color;
    }
}

void MenuFormatColorOnClick(object obj, EventArgs ea)
{
    clrdlg.Color = textbox.BackColor;
    if (clrdlg.ShowDialog() == DialogResult.OK)
    textbox.BackColor = clrdlg.Color;
}

void MenuHelpAboutOnClick(object obj, EventArgs ea)
{
    MessageBox.Show(strProgName + " © 2001 by Charles Petzold", strProgName);
}

protected override void LoadRegistryInfo(RegistryKey regkey)
{
    base.LoadRegistryInfo(regkey);
    textbox.WordWrap = Convert.ToBoolean((int) regkey.GetValue(strWordWrap));
    textbox.Font = new Font((string) regkey.GetValue(strFontFace), float.Parse((string) regkey.GetValue(strFontSize)), (FontStyle) regkey.GetValue(strFontStyle));}
This version of the program also implements a Help menu that includes an About item. I moved the creation of this item to yet another override of the `OnLoad` method. Doing so allows the About item to include the `strProgName` field indicating the name of the program. If the About item were created in the constructor, any program that derived from this one would have the wrong program name in the menu item.

All the options you specify on the Format menu are saved in the registry, so this program again overrides the `LoadRegistryInfo` and `SaveRegistryInfo` methods.

In Chapter 21, I'll enhance the Notepad Clone program by implementing the three items on the File menu involved with printing (in NotepadCloneWithPrinting). In Chapter 24, I'll complete the program by adding drag-and-drop functionality. You'll be able to drag a file or selected text from another application into the program (in the final version named simply NotepadClone).

### Special-Purpose Text Boxes

Although text boxes are used most often to allow a user to enter and edit text, they have a couple special-purpose uses as well, which are indicated by the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code></td>
<td><code>PasswordChar</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>CharacterCasing</code></td>
<td><code>CharacterCasing</code></td>
<td>get/set</td>
</tr>
</tbody>
</table>
You can use a text box in situations where the user must enter a password or other information that should be hidden from others. Normally, `PasswordChar` is 0, but if you set it to something else, everything the user types in the text box will appear as that character. Generally, password boxes use an asterisk for this purpose. The `Multiline` property must be `false` for `PasswordChar` to work.

Sometimes in connection with password entry, text must be converted to uppercase or lowercase. That's the purpose of the `CharacterCasing` property. Set it to one of the following values:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>Upper</td>
<td>1</td>
</tr>
<tr>
<td>Lower</td>
<td>2</td>
</tr>
</tbody>
</table>

The `ReadOnly` property is defined in `TextBoxBase` and also applies to `RichTextBox` controls. If this property is set to `true`, the user can’t type anything into the text box. However, the text box still has a caret, and text can still be selected and copied to the clipboard. A read-only text box is an excellent choice for programs that must display textual information to the user, particularly when the length of the information can’t be anticipated. While a `Label` control might work fine for short strings, the `TextBox` lets the user scroll longer blocks of text. The ability to copy text from the text box to the clipboard is an added bonus. The next program in this chapter (EnumerateEnumeration) uses a read-only text box.

**The Rich Text Box**

I haven't gone into many details about `RichTextBox` because, I'm afraid, the topic is just too big for this book. The class is so named for two reasons. First, it supports rich text, which means formatted text—text that can have a variety of fonts, paragraph indents, and tabs. The second reason is that `RichTextBox` imports and exports text according to the specification known as the Rich Text Format (RTF). RTF was developed at Microsoft around 1986 for the purpose of exchanging formatted text among Windows applications. RTF version 1.6 is documented at [http://msdn.microsoft.com/library/specs/rtfspec.htm](http://msdn.microsoft.com/library/specs/rtfspec.htm).

An RTF document is a text file that defines the back slash (`\`) for formatting tags (such as `\i` to begin a block of italic text) and the curly braces `{ and `}` for enclosing groups of tags. Although RTF has a long history and is supported as an exchange format by many word processors, at this time in the evolution of formatted text, perhaps the biggest problem with RTF is that it's not HTML.

While Notepad is built around the Win32 equivalent of the `TextBox` control, WordPad is built around the Win32 equivalent of the `RichTextBox` control. In a `TextBox` control, you can specify a color or a font for the entire contents of the control; in a `RichTextBox` control, you can specify multiple fonts, colors, paragraph alignments, indents, and so forth. You specify this formatting based on the current selection by using a number of properties, such as `SelectionFont`, `SelectionColor`, `SelectionAlignment`, `SelectionIndent`, and so on.

To get access to the RTF data, use these two properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Rtf</td>
<td>get/set</td>
</tr>
<tr>
<td>string</td>
<td>SelectedRtf</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `RichTextBox` class also includes two methods (with three overloads each) that let you load a file directly into the control (`LoadFile`) and save the document directly to a file (`SaveFile`).
ToolTips

A ToolTip is a small rectangular window that displays some helpful explanatory text, usually when the mouse pointer hovers over a particular control. ToolTips are implemented in Windows Forms in the `ToolTip` class. Although ToolTips in Windows Forms are more limited than those defined in the Win32 API, they are also much easier to use.

You need to define only one ToolTip object to provide ToolTips for a collection of controls:

```csharp
ToolTip tooltip = new ToolTip();
```

Generally, you'll define a ToolTip object for your form (if your form contains controls) and in the constructor of each dialog box your program creates. If your form contains no controls other than a toolbar and a status bar, you don't need a ToolTip object in your form because those controls have their own ToolTip facility. (Chapter 20 covers toolbars and status bars.)

ToolTip is one of those rare classes in which a couple of methods are much more important than its properties and events:

**ToolTip Methods (selection)**

```csharp
void SetToolTip(Control cntl, string strTip)
string GetToolTip(Control cntl)
void RemoveAll()
```

For the single ToolTip object you create for a particular dialog box, you can call `SetToolTip` once for each control to associate ToolTip text with the control, as here:

```csharp
ToolTip tooltip = new ToolTip();
tooltip.SetToolTip(btnBigger, "This button increases the font size");
tooltip.SetToolTip(btnSmaller, "This button decreases the font size");
```

To make the ToolTip text appear in multiple lines, use line feed characters ("\n"). For lengthy ToolTip text, I usually insert line feed characters every 32 characters or so.

If you want to remove a ToolTip for a particular control, set the text to null:

```csharp
tooltip.SetToolTip(btn, null)
```

To remove all the ToolTips, call

```csharp
tooltip.RemoveAll()
```

There's no way to get a list of all the controls for which ToolTips have been defined.

The following two ToolTip properties affect the visibility of the ToolTip text:

**ToolTip Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Active</td>
<td>get/set</td>
<td>True</td>
</tr>
<tr>
<td>bool</td>
<td>ShowAlways</td>
<td>get/set</td>
<td>False</td>
</tr>
</tbody>
</table>

Set `Active` to `false` to disable the display of ToolTip text for all the controls associated with the ToolTip object. Set `ShowAlways` to `true` to display the ToolTip text even if the parent form of the control isn't currently active.

To prevent hectic pop-up activity on the screen, ToolTips usually aren't displayed immediately. A period of time elapses (by default, 0.5 second) after the mouse pointer has stopped moving before the ToolTip is displayed. After another period of time (5 seconds by default), the ToolTip is removed.
from the screen. The following properties control the timing associated with a particular ToolTip object:

**ToolTip Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>AutomaticDelay</code></td>
<td>get/set</td>
<td>500 (milliseconds)</td>
</tr>
<tr>
<td>int</td>
<td><code>InitialDelay</code></td>
<td>get/set</td>
<td><code>AutomaticDelay</code></td>
</tr>
<tr>
<td>int</td>
<td><code>ReshowDelay</code></td>
<td>get/set</td>
<td><code>AutomaticDelay / 5</code></td>
</tr>
<tr>
<td>int</td>
<td><code>AutoPopDelay</code></td>
<td>get/set</td>
<td><code>10 × AutomaticDelay</code></td>
</tr>
</tbody>
</table>

When you set the `AutomaticDelay` property, the other three properties are automatically set to the values indicated in the "Default" column of the table. The idea here is that you can change all timings proportionally just by changing the one property. However, you can then change the other three properties independently. The other three properties are the ones that directly affect the ToolTip display:

- `InitialDelay` indicates the period of time from when the mouse cursor stops moving to when the ToolTip text is displayed.
- The `ReshowDelay` property is the period of time before a new ToolTip is displayed when you move the mouse cursor from one control to another.
- `AutoPopDelay` indicates the time that a ToolTip remains displayed.

Let's take a look at a program that has three single-line text boxes, a multiline read-only text box, and ToolTips to make life easier for the user. Although this program is one of only two examples of ToolTips in this book, you should probably implement them everywhere you use controls.

Throughout this book, I've shown tables of the various enumerations defined in the .NET Framework class library, usually with the numeric values associated with the enumeration members. You may have wondered where these tables came from, since the .NET documentation doesn't include the numeric values. I constructed those tables from a program I wrote early on in my .NET exploration and which I've polished into the following piece of code I call EnumerateEnumeration.

**EnumerateEnumeration.cs**

```csharp
//---------------------------------------------------
// EnumerateEnumeration.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Reflection;           // For the Assembly class
using System.Text;                 // For the StringBuilder class
using System.Windows.Forms;

class EnumerateEnumeration: Form
{
    Button button;
    TextBox tbLibrary, tbNamespace, tbEnumeration, tbOutput;
    CheckBox cbHex;

    public static void Main()
    {
        Application.Run(new EnumerateEnumeration);
    }
}
```
Enumerate Enumeration;

ClientSize = new Size(242, 164);

Label label = new Label();
label.Parent = this;
label.Text = "Library:"
label.Location = new Point(8, 8);
label.Size = new Size(56, 8);

tbLibrary = new TextBox();
tbLibrary.Parent = this;
tbLibrary.Text = "system.windows.forms"
tbLibrary.Location = new Point(64, 8);
tbLibrary.Size = new Size(120, 12);
tbLibrary.Anchor |= AnchorStyles.Right;

ToolTip tooltip = new ToolTip();
tooltip.SetToolTip(tbLibrary,
    "Enter the name of a .NET dynamic-\n" +
    "link library, such as 'mscorlib',\n" +
    "'system.windows.forms', or\n" +
    "'system.drawing'."];

label = new Label();
label.Parent = this;
label.Text = "Namespace:"
label.Location = new Point(8, 24);
label.Size = new Size(56, 8);

tbNamespace = new TextBox();
tbNamespace.Parent = this;
tbNamespace.Location = new Point(64, 24);
tbNamespace.Size = new Size(120, 12);
tbNamespace.Anchor |= AnchorStyles.Right;

tooltip.SetToolTip(tbNamespace,
    "Enter the name of a namespace\n" +
    "within the library, such as\n" +
    "'System', 'System.IO',\n" +
    "'System.Drawing',\n" +
    "'System.Drawing.Drawing2D',\n" +
    "or 'System.Windows.Forms'."];
label = new Label();
label.Parent = this;
label.Text = "Enumeration:";
label.Location = new Point(8, 40);
label.Size = new Size(56, 8);

tbEnumeration = new TextBox();
tbEnumeration.Parent = this;
tbEnumeration.Text = "ScrollBars";
tbEnumeration.Location = new Point(64, 40);
tbEnumeration.Size = new Size(120, 12);
tbEnumeration.Anchor |= AnchorStyles.Right;

tooltip.SetToolTip(tbEnumeration,
"Enter the name of an enumeration\n" +
"defined in the namespace.");

cbHex = new CheckBox();
cbHex.Parent = this;
cbHex.Text = "Hex";
cbHex.Location = new Point(192, 16);
cbHex.Size = new Size(40, 8);
cbHex.Anchor = AnchorStyles.Top | AnchorStyles.Right;
cbHex.CheckedChanged += new EventHandler(CheckBoxOnCheckedChanged);

tooltip.SetToolTip(cbHex, "Check this box to display the\n" +
"enumeration values in hexadecimal.");

button = new Button();
button.Parent = this;
button.Text = "OK";
button.Location = new Point(192, 32);
button.Size = new Size(40, 16);
button.Anchor = AnchorStyles.Top | AnchorStyles.Right;
button.Click += new EventHandler(ButtonOkOnClick);

AcceptButton = button;

tooltip.SetToolTip(button,
"Click this button to display results.");

tbOutput = new TextBox();
tbOutput.Parent = this;
tbOutput.ReadOnly = true;
tbOutput.Multiline = true;
tbOutput.ScrollBars = ScrollBars.Vertical;
tbOutput.Location = new Point(8, 56);
tbOutput.Size = new Size(226, 100);
tbOutput.Anchor = AnchorStyles.Left | AnchorStyles.Top |
AnchorStyles.Right | AnchorStyles.Bottom;

AutoSizeBaseSize = new Size(4, 8);

// Initialize the display.

ButtonOkOnClick(button, EventArgs.Empty);
}

void CheckBoxOnCheckedChanged(object sender, EventArgs ea)
{
    button.PerformClick();
}

void ButtonOkOnClick(object sender, EventArgs ea)
{
    FillTextBox(tbOutput, tbLibrary.Text, tbNamespace.Text,
    tbEnumeration.Text, cbHex.Checked);
}

public static bool FillTextBox(TextBox tbOutput, string strLibrary,
    string strNamespace,
    string strEnumeration, bool
    bHexadecimal)
{
    string strEnumText = strNamespace + "." + strEnumeration;
    string strAssembly;

    try
    {
        strAssembly =
            Assembly.LoadWithPartialName(strLibrary).FullName;
    }
    catch
    {
        return false;
    }
    string strFullText = strEnumText + "," + strAssembly;

    // Get the type of the enum.

    Type type = Type.GetType(strFullText, false, true);

    if(type == null)
{  
    tbOutput.Text = "" + strFullText +  
        "\n is not a valid type.";
    return false;
}
else if(!type.IsEnum)
{
    tbOutput.Text = "" + strEnumText +  
        "\n is a valid type but not an enum.";
    return false;
}

// Get all the members in that enum.

string[] astrMembers = Enum.GetNames(type);
Array arr = Enum.GetValues(type);
object[] aobjMembers = new object[arr.Length];

arr.CopyTo(aobjMembers, 0);

    // Create a StringBuilder for the text.

StringBuilder sb = new StringBuilder();

    // Append the enumeration name and headings.

sb.Append(strEnumeration);  
sb.Append(" Enumeration\r\nMember	Value\r
");

    // Append the text rendition and the actual numeric values.

for (int i = 0; i < astrMembers.Length; i++)
{
    sb.Append(astrMembers[i]);
    sb.Append("\t");

    if (bHexadecimal)
        sb.Append("0x" + Enum.Format(type, aobjMembers[i], "X"));
    else
        sb.Append(Enum.Format(type, aobjMembers[i], "D"));
    sb.Append("\r\n");
}

    // Append some other information.

The constructor creates three pairs of labels and single-line text boxes to let the user type in a DLL name, a namespace in that library, and an enumeration in that namespace. A check box lets you indicate that the results should be displayed in hexadecimal. A push button lets you indicate when everything is finished and the results should be displayed. Each of these controls is associated with some ToolTip text.

The results are displayed in a read-only text box. Notice the use of the Anchor property to make all the text boxes flexible in size. As you make the form wider, all three single-line text boxes increase in width. As you make the form shorter or taller, the read-only text box changes height.

Here's a view of the program from the last time I used it:

![Image](image.jpg)

Notice that the read-only text box has a different default background color than the normal text boxes. Because read-only text boxes still implement a caret and clipboard interface, I've been able to select the text I want from the text box and type Ctrl+C to copy it to the clipboard. I then paste the text into my Microsoft Word document for the chapter and convert the text to a Word table.

After a little while working with this program, you may begin to be annoyed at having to retype various commonly used libraries and namespaces. Later on in this chapter, I'll present another version of this program, named EnumerateEnumerationCombo, that has combo boxes that use the Windows registry to save all valid combinations of library names, namespaces, and enumerations.

To accommodate this second program is the reason that all the display code is isolated in the FillTextBox method, which I've also defined as static. It's static because EnumerateEnumerationCombo needs to make use of the FillTextBox method but doesn't derive from EnumerateEnumeration. FillTextBox returns true if the combination of three text strings was valid and false otherwise. EnumerateEnumeration doesn't use this information; EnumerateEnumerationCombo does.

FillTextBox uses the GetType method of the Type class to obtain a Type object for the enumeration. The argument to GetType is a text string that takes the following form:

namespace.enumeration.library
Notice the normal period separating the namespace and the enumeration name, and also the comma preceding the library name. The library name must include version information, which is the reason for the `Assembly.LoadWithPartialName` call.

The program obtains the enumeration member names and values from the static `GetNames` and `GetValues` methods of the `Enum` class.

The relationship between the library name and the namespace can be a bit tricky: The `system.drawing.dll` library contains the namespace `System.Drawing`. However, `System.Drawing.Drawing2D` is also located in `system.drawing.dll`. Many of the basic namespaces (such as `System` and `System.IO`) are located in `mscorlib.dll`, which stands for "Microsoft Core Library."

Notice the use of the `StringBuilder` class to build the string that’s displayed in the read-only text box. (I discuss `StringBuilder` in Appendix C.) The original version put everything into the text box line-by-line using string appending with the `+=` operator. I was beginning to suspect a problem with that approach when the `EmfPlusRecordType` enumeration required 30 seconds to display using a pre-release version of Visual Studio .NET and my pokey machine. Switching to the `StringBuilder` class made the update instantaneous.

**The List Box**

The `ListBox` control is often used in a manner similar to a group of radio buttons—to provide a way for a user to pick one item from a list of several items. However, list boxes generally take up less space on the screen and also let the user select multiple items. The `ComboBox` control (which I'll talk about later in this chapter) usually takes up even less space and often includes an area for the user to type information, much like a `TextBox` control.

Both `ListBox` and `ComboBox` are derived from `ListControl`, as shown in the following class hierarchy:

![Class Hierarchy Diagram]

Usually after creating a list box, you want to fill it with items. You do that using the `Items` property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListBox.ObjectCollection</td>
<td>Items</td>
<td>get</td>
</tr>
</tbody>
</table>

The `ObjectCollection` class is by now a familiar implementation of the `IList`, `ICollection`, and `IEnumerable` interfaces. You can index `Items` as an array, use the `Add` and `AddRange` methods to add items to the list box, and search for items with the `Contains` and `IndexOf` methods.

The items in a `ListBox` control are defined to be of type `Object`, not necessarily strings. Each object in `ListBox` is displayed using the object's `ToString` method. Of course, when the program retrieves items from the list box, it is responsible for casting the object to its proper type.

It's possible for the list box to contain multiple identical items. For example, if you add an item to a list box that's already in the collection, the collection will contain two copies of the item. Because such a situation confuses the user, it's probably undesirable.
As you add items to a list box, each item is assigned an index beginning at 0. The index determines the item's position in the list box. Generally, the indices are consecutive as the items are added. However, if you've set the Sorted property to true, the indices will be consecutive based on the alphabetical order of the items.

**ListBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Sorted</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>TopIndex</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The TopIndex property indicates the index of the item displayed at the top of the list box. By default, TopIndex is 0.

The PreferredHeight property indicates the height of the list box required to fit all the items:

**ListBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>PreferredHeight</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IntegralHeight</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Generally, you don't set the height of the list box to PreferredHeight. The IntegralHeight property is true by default to adjust the height you set so that partial items aren't displayed.

If the height of the list box doesn't accommodate the full number of items, a vertical scroll bar will be displayed. Optionally, you can display the scroll bar regardless of the number of items. You can also optionally display a horizontal scroll bar for items that exceed the width of the list box:

**ListBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>ScrollAlwaysVisible</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>HorizontalScrollbar</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The following properties of ListBox involve the selection of items:

**ListBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionMode</td>
<td>SelectionMode</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>SelectedIndex</td>
<td>get/set</td>
</tr>
<tr>
<td>object</td>
<td>SelectedItem</td>
<td>get</td>
</tr>
<tr>
<td>ListBox.SelectedIndexCollection</td>
<td>SelectedIndices</td>
<td>get</td>
</tr>
<tr>
<td>ListBox.SelectedObjectCollection</td>
<td>SelectedItems</td>
<td>get</td>
</tr>
</tbody>
</table>

The SelectionMode property is one of the following members of the SelectionMode enumeration:

**SelectionMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>One</td>
<td>1</td>
</tr>
<tr>
<td>MultiSimple</td>
<td>2</td>
</tr>
<tr>
<td>MultiExtended</td>
<td>3</td>
</tr>
</tbody>
</table>
The default is One. At any time, only one item in the list box is highlighted, which is indicated by a reverse-video display. Whenever the list box has the input focus, the same item is also surrounded by a dotted line—a focus rectangle similar to that on push buttons—but the focus rectangle might be a bit difficult to see because it's the same size as the reverse-video rectangle. You can also select an item with the mouse.

By default, a newly created and filled list box has no selection, in which case SelectedIndex returns −1 and SelectedItem returns null. You'll probably want to use one of these two properties to initialize the list box to a particular index or item.

With the MultiSimple option, the user can select multiple items in the list box. Each selected item is indicated by reverse-video. The focus rectangle is independent of any selection rectangle. You can move the dotted-line focus rectangle among the items using the cursor-movement keys. Pressing the spacebar selects (or deselects) the item indicated by the focus rectangle. In addition, you can select (or deselect) an item with the mouse, in which case that item also gets the focus rectangle.

A list box with the MultiExtended option at first looks much like a single-selection list box. Using the cursor-movement keys, you change both the reverse-video selection rectangle and the dotted-line focus rectangle. However, you can extend a selection by holding down the Shift key while pressing the cursor-movement keys. But if you then release the Shift key and press a cursor-movement key—or click an item with the mouse—the selection again becomes just one item. In a MultiExtended list box, you can also select (or deselect) individual items by clicking them with the mouse while the Ctrl key is pressed.

The MultiExtended list box probably makes most sense when the user is likely to select a range of items. Use MultiSimple when the items the user is likely to select are not consecutive.

In a single-selection list box, use SelectedIndex or SelectedItem to obtain the selected index or item. If lstbox is an object of type ListBox,

```csharp
lstbox.SelectedItem
```

is equivalent to

```csharp
lstbox.Items[lstbox.SelectedIndex]
```

You can also use the Text property to obtain the text representation of the selected item. The expression

```csharp
lstbox.Text
```

is equivalent to

```csharp
lstbox.SelectedItem.ToString()
```

In a multiselection list box, the properties SelectedIndices and SelectedItems give you access to the selected items. The SelectedIndexCollection and SelectedObjectCollection classes both have Count properties and indexers. The Count property indicates the number of selected items, and—as usual—the indexer can range from 0 to (Count−1). The indexer for the SelectedIndices object returns the index of the selected item within the list box. For example, if index is a number between 0 and (Count−1),

```csharp
lstbox.SelectedItems[index]
```

is equivalent to

```csharp
lstbox.Items[lstbox.SelectedIndices[index]]
```

To initialize a multiselection list box, you can call the SetSelected method for each item you want to select:

**ListBox Methods (selection)**

```csharp
void SetSelected(int index, bool bSelect)
bool GetSelected(int index)
void ClearSelected()
```
In addition, the `GetSelected` method returns `true` for each index that corresponds to a selected item. `ClearSelected` deselects all items in the list box.

Often when you use list boxes in a dialog box, you need only obtain the selected item or items when the user presses the OK button. However, at times, you'll want to react whenever the selected item changes. For that purpose there are two events:

**ListBox Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectedIndexChanged</td>
<td>OnSelectedIndexChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>SelectedValueChanged</td>
<td>OnSelectedValueChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

`SelectedValueChanged` is implemented in `ListControl` and is also available in `ComboBox`. These two events are basically equivalent and occur whether the selection changes programmatically or by the user. In a `MultiSimple` list box, these two events are fired even when the focus rectangle changes and not the selection.

Here's a simple program that lists the current MS-DOS environment variable names in a list box and displays the value of the currently selected item.

**EnvironmentVars.cs**

```csharp
using System;
using System.Collections;
using System.Drawing;
using System.Windows.Forms;

class EnvironmentVars: Form
{
    Label label;

    public static void Main()
    {
        Application.Run(new EnvironmentVars());
    }

    public EnvironmentVars()
    {
        Text = "Environment Variables";

        // Create Label control.
        label = new Label();
        label.Parent   = this;
        label.Anchor   = AnchorStyles.Left | AnchorStyles.Right;
        label.Location = new Point(Font.Height, Font.Height);
        label.Size     = new Size(ClientSize.Width - 2 * Font.Height,
```

// Create ListBox control.

ListBox listbox = new ListBox();
listbox.Parent = this;
listbox.Location = new Point(Font.Height, 3 * Font.Height);
listbox.Size = new Size(12 * Font.Height, 8 * Font.Height);
listbox.Sorted = true;
listbox.SelectedIndexChanged +=
    new EventHandler(ListBoxOnSelectedIndexChanged);

// Set environment strings in ListBox control.

IDictionary dict = Environment.GetEnvironmentVariables();
string[] astr = new String[dict.Keys.Count];
dict.Keys.CopyTo(astr, 0);
listbox.Items.AddRange(astr);
listbox.SelectedIndex = 0;
}

void ListBoxOnSelectedIndexChanged(object obj, EventArgs ea)
{
    ListBox listbox = (ListBox) obj;
    string strItem = (string) listbox.SelectedItem;

    label.Text = Environment.GetEnvironmentVariable(strItem);
}

Notice that the environment names are an array of strings added to the list box using the AddRange method. The constructor concludes by setting the SelectedIndex property to 0. Doing so generates a call to the SelectedIndexChanged event handler, which retrieves the selected item and sets the Label text with it. Here's a sample view of the program:
List boxes have an owner-draw option that you can enable by setting the `DrawMode` property. You set the property to one of the members of the `DrawMode` enumeration, which specifies whether all the items have the same height or are different heights:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>OwnerDrawFixed</td>
<td>1</td>
</tr>
<tr>
<td>OwnerDrawVariable</td>
<td>2</td>
</tr>
</tbody>
</table>

As usual, you must also install handlers for the `MeasureItem` and `DrawItem` events.

**List Box + Text Box = Combo Box**

In its classical form, the combo box looks like a text box with an arrow at the right. Click the arrow and a list box drops down. But this traditional usage is not the only way in which you can use combo boxes. The critical property is `DropDownStyle`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComboBoxStyle</td>
<td>DropDownStyle</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>DroppedDown</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `DropDownStyle` property is set to a member of the `ComboBoxStyle` enumeration:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>0</td>
<td>Editable field, list always present</td>
</tr>
<tr>
<td>DropDown</td>
<td>1</td>
<td>Editable field, list drops down (default)</td>
</tr>
<tr>
<td>DropDownList</td>
<td>2</td>
<td>Noneditable field, list drops down</td>
</tr>
</tbody>
</table>

What I referred to as the classical combo box has the default style of DropDown. The user can type something into the edit field or select something from the list. The style of DropDownList is most like a regular list box except it takes up less space. The `DroppedDown` property is true if the list part of the combo box is visible.

Like a list box, a combo box has an `Items` property that contains all the items in the list:
**ComboBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComboBox.ObjectCollection</td>
<td>Items</td>
<td>get</td>
</tr>
</tbody>
</table>

The `ObjectCollection` class has a `Count` property, an indexer, and familiar methods such as `Add`, `AddRange`, `Insert`, and `Remove`.

The following properties indicate the index of the selected item and the selected item itself:

**ComboBox Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>SelectedIndex</td>
<td>get/set</td>
</tr>
<tr>
<td>object</td>
<td>SelectedItem</td>
<td>get/set</td>
</tr>
</tbody>
</table>

In a `ListBox` object, `SelectedIndex` and `SelectedItem` are usually valid, except possibly after a list box is first created and the properties aren't initialized. In a `ComboBox`, however, if the user is currently typing something into the edit field, `SelectedIndex` returns –1 and `SelectedItem` returns `null`.

The `Text` property always indicates the text that appears in the edit field. Thus, the `Text` property changes as the user selects different items from the list part of the combo box and as the user types something into the edit field.

The following table shows the most useful events implemented by `ComboBox`:

**ComboBox Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextChanged</td>
<td>OnTextChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>SelectedIndexChanged</td>
<td>OnSelectedIndexChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>SelectionChangeCommitted</td>
<td>OnSelectionChangeCommitted</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

The `TextChanged` event occurs when the user scrolls through the items in the list box or types something into the edit field. The `SelectedIndexChanged` event occurs only when the user scrolls through the items in the list box. For `DropDown` and `DropDownList` combo boxes, if the list part is dropped down and the user scrolls through the items, both `TextChanged` and `SelectedIndexChanged` events occur; only when the drop-down list is retracted does the `SelectionChangeCommitted` event occur. However, if the user scrolls through items using the cursor keys without causing the drop-down list to drop down, `SelectionChangeCommitted` events occur with each change in the selection. For `Simple` combo boxes, `SelectionChangeCommitted` events occur whenever the selection changes.

In the following enhancement of the `EnumerateEnumeration` program, I replaced all three list boxes with combo boxes and installed `TextChanged` event handlers for all three. By handling `TextChanged` events, the program can test for valid combinations with every keystroke that the user types into the edit field. The push button is eliminated because it is no longer necessary.

**EnumerateEnumerationCombo.cs**

```csharp
// EnumerateEnumerationCombo.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;
```

```csharp
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;
```
class EnumerateEnumerationCombo: Form
{
    CheckBox cbHex;
    ComboBox comboLibrary, comboNamespace, comboEnumeration;
    TextBox tbOutput;

    const string strRegKeyBase = "Software\ProgrammingWindowsWithCSharp\EnumerateEnumerationCombo";

    public static void Main()
    {
        Application.Run(new EnumerateEnumerationCombo());
    }

    public EnumerateEnumerationCombo()
    {
        Text = "Enumerate Enumeration (Combo)";
        ClientSize = new Size(242, 164);

        Label label = new Label();
        label.Parent = this;
        label.Text = "Library:";
        label.Location = new Point(8, 8);
        label.Size = new Size(56, 8);

        comboLibrary        = new ComboBox();
        comboLibrary.Parent = this;
        comboLibrary.DropDownStyle = ComboBoxStyle.DropDown;
        comboLibrary.Sorted = true;
        comboLibrary.Location = new Point(64, 8);
        comboLibrary.Size = new Size(120, 12);
        comboLibrary.Anchor |= AnchorStyles.Right;
        comboLibrary.TextChanged += new EventHandler(ComboBoxLibraryOnTextChanged);

        label          = new Label();
        label.Parent   = this;
        label.Text     = "Namespace:";
        label.Location = new Point(8, 24);
        label.Size     = new Size(56, 8);

        comboNamespace               = new ComboBox();
        comboNamespace.Parent        = this;
        comboNamespace.DropDownStyle = ComboBoxStyle.DropDown;
        comboNamespace.Sorted        = true;
        comboNamespace.Location      = new Point(64, 24);
        comboNamespace.Size          = new Size(120, 12);
        comboNamespace.Anchor       |= AnchorStyles.Right;
        comboNamespace.TextChanged  += new EventHandler(ComboBoxNamespaceOnTextChanged);

    }
comboNamespace.Location = new Point(64, 24);
comboNamespace.Size = new Size(120, 12);
comboNamespace.Anchor |= AnchorStyles.Right;
comboNamespace.TextChanged +=
    new EventHandler(ComboBoxNamespaceOnTextChanged);

label = new Label();
label.Parent = this;
label.Text = "Enumeration:";
label.Location = new Point(8, 40);
label.Size = new Size(56, 8);

comboEnumeration = new ComboBox();
comboEnumeration.Parent = this;
comboEnumeration.DropDownStyle = ComboBoxStyle.DropDown;
comboEnumeration.Sorted = true;
comboEnumeration.Location = new Point(64, 40);
comboEnumeration.Size = new Size(120, 12);
comboEnumeration.Anchor |= AnchorStyles.Right;
comboEnumeration.TextChanged +=
    new EventHandler(ComboBoxEnumerationOnTextChanged);

cbHex = new CheckBox();
cbHex.Parent = this;
cbHex.Text = "Hex";
cbHex.Location = new Point(192, 25);
cbHex.Size = new Size(40, 8);
cbHex.Anchor = AnchorStyles.Top | AnchorStyles.Right;
cbHex.CheckedChanged += new EventHandler(CheckBoxOnCheckedChanged);

tbOutput = new TextBox();
tbOutput.Parent = this;
tbOutput.ReadOnly = true;
tbOutput.Multiline = true;
tbOutput.ScrollBars = ScrollBars.Vertical;
tbOutput.Location = new Point(8, 56);
tbOutput.Size = new Size(226, 100);
tbOutput.Anchor = AnchorStyles.Left | AnchorStyles.Top | 
    AnchorStyles.Right | AnchorStyles.Bottom;

AutoScaleBaseSize = new Size(4, 8);

// Initialize display.
FillComboBox(comboLibrary, strRegKeyBase);
UpdateTextBox();

void ComboBoxLibraryOnTextChanged(object obj, EventArgs ea)
{
    FillComboBox(comboNamespace, strRegKeyBase + "\" +
                  comboLibrary.Text);
    ComboBoxNamespaceOnTextChanged(obj, ea);
}
void ComboBoxNamespaceOnTextChanged(object obj, EventArgs ea)
{
    FillComboBox(comboEnumeration, strRegKeyBase + "\" +
                 comboLibrary.Text + "\" +
                 comboNamespace.Text);
    ComboBoxEnumerationOnTextChanged(obj, ea);
}
void ComboBoxEnumerationOnTextChanged(object obj, EventArgs ea)
{
    UpdateTextBox();
}
void CheckBoxOnCheckedChanged(object obj, EventArgs ea)
{
    UpdateTextBox();
}
void UpdateTextBox()
{
    if (EnumerateEnumeration.FillTextBox(tbOutput, comboLibrary.Text,
                                          comboNamespace.Text, comboEnumeration.Text, cbHex.Checked))
    {
        if (!comboLibrary.Items.Contains(comboLibrary.Text))
            comboLibrary.Items.Add(comboLibrary.Text);
        if (!comboNamespace.Items.Contains(comboNamespace.Text))
            comboNamespace.Items.Add(comboNamespace.Text);
        if (!comboEnumeration.Items.Contains(comboEnumeration.Text))
            comboEnumeration.Items.Add(comboEnumeration.Text);

        string strRegKey = strRegKeyBase + "\" +
                           comboLibrary.Text + "\" +
                           comboNamespace.Text + "\" +
                           comboEnumeration.Text;
When you first run the program, all three combo boxes are empty. It is your responsibility to type a valid library, namespace, and enumeration name. Like I said, the program checks for a valid combination with every keystroke. When a valid combination is encountered, the enumeration information is displayed by the static FillTextBox method in the original EnumerateEnumeration program.

The program also stores the valid combination in the registry, and each valid combination is then added to the combo boxes. The tree structure of the registry is perfect for an application like this. If you look in the Registry Editor after entering a few valid combinations of libraries, namespaces, and enumerations, you’ll see that no actual data is stored in the registry! Each valid combination becomes a key. You’ll see entries with keys like this:

Software\ProgrammingWindowsWithCSharp\EnumerateEnumerationCombo\system.drawing\System.Drawing.Drawing2D\DashStyle

And this:

Software\ProgrammingWindowsWithCSharp\EnumerateEnumerationCombo\system.windows.forms\System.Windows.Forms\DockStyle
As you change the selection or text in the first combo box (where you enter the library name), the `FillComboBox` method in the program obtains the list of registry subkeys of the particular library name. Those subkeys are used to fill the second combo box (the one for the namespace names). Similarly, as you change the selection or text in the namespace combo box, `FillComboBox` obtains the list of valid enumerations for the third combo box. The more you use the program, the more useful it becomes.

**Up-Down Controls**

The Windows Forms up-down control is more traditionally known as a *spin* control. There are two types of up-down controls, as shown in the following class hierarchy:

![Class Hierarchy Diagram]

The controls consist of an edit field with a pair of arrow buttons at the right. The `NumericUpDown` control lets the user select from a range of numbers while the `DomainUpDown` control lets the user select from a collection of objects that are identified by strings. I'll be focusing on the `NumericUpDown` control in this section.

You generally set the following properties to initialize the control:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decimal</code></td>
<td><code>Value</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>decimal</code></td>
<td><code>Minimum</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>decimal</code></td>
<td><code>Maximum</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>decimal</code></td>
<td><code>Increment</code></td>
<td>get/set</td>
</tr>
</tbody>
</table>

Notice the `decimal` type. For more information on `decimal`, see Appendix B. The `Increment` property indicates the change in the `Value` when the up and down arrows are clicked. The defaults let `Value` range from 0 through 100 with an increment of 1. You can set the `Minimum` and `Maximum` properties to `Decimal.MinValue` and `Decimal.MaxValue` to effectively remove any limitations.

The `Minimum` and `Maximum` properties are very strict: if the user manually enters a number outside the `Minimum` and `Maximum` range, the spin control changes the number to either `Minimum` or `Maximum`. (Watch out for this in modal dialog boxes. It's possible the OK button will dismiss the dialog box before the user has a chance to notice that the value has changed.) If the program sets the `Value` property to a number outside the range, an exception is raised.

The following properties control the display of the number in the control:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>DecimalPlaces</code></td>
<td>get/set</td>
<td>0</td>
</tr>
</tbody>
</table>
The `ValueChanged` event indicates when the value of the control has changed, either by the user or by the program:

**NumericUpDown Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>ThousandsSeparator</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>Hexadecimal</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>

**NumericUpDown Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueChanged</td>
<td>OnValueChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

To demonstrate `NumericUpDown` controls, I’ve written a program that uses nine of them. The Transform program lets you experiment with matrix transforms, either by altering the six elements of the matrix or by effectively making calls to the various methods implemented by the `Matrix` class, such as `Scale` or `Shear`. I’ve divided the program into three files, one for the form and the other two for the two dialog boxes.

The bulk of the main form consists of two methods called during the `Paint` event: `DrawAxes` draws a coordinate system, and `DrawHouse` draws a little house. The house is drawn based on a `Matrix` object stored as a field.

**Transform.cs**

```csharp
// Transform.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;      // For bitmap
using System.Windows.Forms;

class Transform: Form
{
    Matrix matrix = new Matrix();

    public static void Main()
    {
        Application.Run(new Transform());
    }
    public Transform()
    {
        Text = "Transform";
        ResizeRedraw = true;
        BackColor = Color.White;
        Size += Size;

        // Create modal dialog box.
    }
}
```
MatrixElements dlg = new MatrixElements();
dlg.Owner = this;
dlg.Matrix = matrix;
dlg.Changed += new EventHandler(MatrixDialogOnChanged);
dlg.Show();
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    
    DrawAxes(grfx);
    grfx.Transform = matrix;
    DrawHouse(grfx);
}

void DrawAxes(Graphics grfx)
{
    Brush brush = Brushes.Black;
    Pen pen = Pens.Black;
    StringFormat strfmt = new StringFormat();

    // Horizontal axis
    strfmt.Alignment = StringAlignment.Center;
    for (int i = 1; i <= 10; i++)
    {
        grfx.DrawLine(pen, 100 * i, 0, 100 * i, 10);
        grfx.DrawString((i * 100).ToString(), Font, brush,
                          100 * i, 10, strfmt);
        grfx.DrawLine(pen, 100 * i, 10 + Font.Height,
                          100 * i, ClientSize.Height);
    }

    // Vertical axis
    strfmt.Alignment = StringAlignment.Near;
    strfmt.LineAlignment = StringAlignment.Center;
    for (int i = 1; i <= 10; i++)
    {
        grfx.DrawLine(pen, 0, 100 * i, 10, 100 * i);
        grfx.DrawString((i * 100).ToString(), Font, brush, 10,
                         100 * i, strfmt);
        float cxText = grfx.MeasureString((i * 100).ToString(), Font).Width;
void DrawHouse(Graphics grfx)
{
    Rectangle rectFacade = new Rectangle(0, 40, 100, 60);
    Rectangle rectDoor = new Rectangle(10, 50, 25, 50);
    Rectangle[] rectWindows = {
        new Rectangle(50, 50, 10, 10),
        new Rectangle(60, 50, 10, 10),
        new Rectangle(70, 50, 10, 10),
        new Rectangle(50, 60, 10, 10),
        new Rectangle(60, 60, 10, 10),
        new Rectangle(70, 60, 10, 10),
        new Rectangle(15, 60, 5, 7),
        new Rectangle(20, 60, 5, 7),
        new Rectangle(25, 60, 5, 7)
    };
    Rectangle rectChimney = new Rectangle(80, 5, 10, 35);
    Point[] ptRoof = {
        new Point(50, 0),
        new Point(0, 40),
        new Point(100, 40)
    };

    // Create bitmap and brush for chimney.
    Bitmap bitmap = new Bitmap(8, 6);
    byte[] bits = {
        0, 0, 0, 0, 0, 0, 0, 0,
        1, 1, 1, 0, 1, 1, 1, 0,
        1, 1, 1, 0, 1, 1, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        1, 0, 1, 1, 1, 0, 1, 1,
        1, 0, 1, 1, 1, 0, 1, 1,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 1, 1};
    for (int i = 0; i < 48; i++)
        bitmap.SetPixel(i % 8, i / 8, bits[i] == 1 ? Color.DarkGray : Color.LightGray);
    Brush brush = new TextureBrush(bitmap);

    // Draw entire house.
    grfx.FillRectangle( Brushes.LightGray, rectFacade );
The program also displays a modeless dialog box titled Matrix Elements, which is implemented in the following source code file. The dialog box has six `NumericUpDown` controls that let you select the six elements of the matrix and also implements an event named `Changed`. When you click the Update button, the `Changed` event is triggered to let the main program know that a new matrix is available. (The main program processes this event in the `MatrixDialogOnChanged` event handler.) The `Matrix` object is accessed as a property.

### MatrixElements.cs

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixElements : Form {
    Matrix          matrix;
    Button          btnUpdate;
    NumericUpDown[] updown = new NumericUpDown[6];

    public event EventHandler Changed;

    public MatrixElements()
    {
    
    }
}
```

The program also displays a modeless dialog box titled Matrix Elements, which is implemented in the following source code file. The dialog box has six `NumericUpDown` controls that let you select the six elements of the matrix and also implements an event named `Changed`. When you click the Update button, the `Changed` event is triggered to let the main program know that a new matrix is available. (The main program processes this event in the `MatrixDialogOnChanged` event handler.) The `Matrix` object is accessed as a property.

### MatrixElements.cs

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixElements : Form {
    Matrix          matrix;
    Button          btnUpdate;
    NumericUpDown[] updown = new NumericUpDown[6];

    public event EventHandler Changed;

    public MatrixElements()
    {
    
    }
}
```
Text = "Matrix Elements";
FormBorderStyle = FormBorderStyle.FixedDialog;
ControlBox = false;
MinimizeBox = false;
MaximizeBox = false;
ShowInTaskbar = false;

String[] strLabel = { "X Scale:", "Y Shear:",
                     "X Shear:", "Y Scale:",
                     "X Translate:", "Y Translate:" };  

for (int i = 0; i < 6; i++)
{
    Label label = new Label();
    label.Parent = this;
    label.Text = strLabel[i];
    label.Location = new Point(8, 8 + 16 * i);
    label.Size = new Size(64, 8);

    updown[i] = new NumericUpDown();
    updown[i].Parent = this;
    updown[i].Location = new Point(76, 8 + 16 * i);
    updown[i].Size = new Size(48, 12);
    updown[i].TextAlign = HorizontalAlignment.Right;
    updown[i].ValueChanged += new EventHandler
                               (UpDownOnValueChanged);
    updown[i].DecimalPlaces = 2;
    updown[i].Increment = 0.1m;
    updown[i].Minimum = Decimal.MinValue;
    updown[i].Maximum = Decimal.MaxValue;
}

btnUpdate = new Button();
btnUpdate.Parent = this;
btnUpdate.Text = "Update";
btnUpdate.Location = new Point(8, 108);
btnUpdate.Size = new Size(50, 16);
btnUpdate.Click += new EventHandler(ButtonUpdateOnClick);

AcceptButton = btnUpdate;

Button btn = new Button();
btn.Parent = this;
btn.Text = "Methods...";
btn.Location = new Point(76, 108);
btn.Size = new Size(50, 16);
btn.Click += new EventHandler(ButtonMethodsOnClick);

ClientSize = new Size(134, 132);

AutoSizeBaseSize = new Size(4, 8);
}

public Matrix Matrix
{
    get
    {
        matrix = new Matrix((float) updown[0].Value,
            (float) updown[1].Value,
            (float) updown[2].Value,
            (float) updown[3].Value,
            (float) updown[4].Value,
            (float) updown[5].Value);

        return matrix;
    }
    set
    {
        matrix = value;

        for (int i = 0; i < 6; i++)
            updown[i].Value = (decimal) value.Elements[i];
    }
}

void UpDownOnValueChanged(object obj, EventArgs ea)
{
    Graphics grfx = CreateGraphics();

    bool boolEnableButton = true;

    try
    {
        grfx.Transform = Matrix;
    }
    catch
    {
        boolEnableButton = false;
    }

    btnUpdate.Enabled = boolEnableButton;
    grfx.Dispose();
}

void ButtonUpdateOnClick(object obj, EventArgs ea)
{

if (Changed != null)
    Changed(this, new EventArgs());
}
void ButtonMethodsOnClick(object obj, EventArgs ea)
{
    MatrixMethods dlg = new MatrixMethods();

    dlg.Matrix = Matrix;

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        Matrix = dlg.Matrix;
        btnUpdate.PerformClick();
    }
}
}

The Matrix Elements dialog box also contains a second button, labeled Methods. That button invokes a modal dialog box titled Matrix Methods, which is implemented in the following source code file. Matrix Methods contains another three `NumericUpDown` controls that provide arguments to the various methods of the `Matrix` class that alter the matrix elements. Each method is implemented by a button that also dismisses the dialog box.

MatrixMethods.cs
//-----------------------------
// MatrixMethods.cs © 2001 by Charles Petzold
//-----------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class MatrixMethods: Form
{
    Matrix          matrix;
    Button          btnInvert;
    NumericUpDown[] updown = new NumericUpDown[3];
    RadioButton[]   radio = new RadioButton[2];

    public MatrixMethods()
    {
        Text = "Matrix Methods";
        FormBorderStyle = FormBorderStyle.FixedSingle;
        ControlBox      = false;
        MinimizeBox     = false;
        MaximizeBox     = false;
        ShowInTaskbar   = false;
        Location        = ActiveForm.Location +
SystemInformation.CaptionButtonSize +
SystemInformation.FrameBorderSize;

String[] astrLabel = {"X / DX:", "Y / DY:", "Angle:"};

for (int i = 0; i < 3; i++)
{
    Label label = new Label();
    label.Parent = this;
    label.Text = astrLabel[i];
    label.Location = new Point(8, 8 + 16 * i);
    label.Size = new Size(32, 8);

    updown[i] = new NumericUpDown();
    updown[i].Parent = this;
    updown[i].Location = new Point(40, 8 + 16 * i);
    updown[i].Size = new Size(48, 12);
    updown[i].TextAlign = HorizontalAlignment.Right;

    updown[i].DecimalPlaces = 2;
    updown[i].Increment = 0.1m;
    updown[i].Minimum = Decimal.MinValue;
    updown[i].Maximum = Decimal.MaxValue;
}

// Create group box and radio buttons.

GroupBox grpbox = new GroupBox();
grpbox.Parent = this;
grpbox.Text = "Order";
grpbox.Location = new Point(8, 60);
grpbox.Size = new Size(80, 32);

for (int i = 0; i < 2; i++)
{
    RadioButton radio = new RadioButton();
    radio.Parent = grpbox;
    radio.Text = new string[] {"Prepend", "Append"}[i];
    radio.Location = new Point(8, 8 + 12 * i);
    radio.Size = new Size(50, 10);
    radio.Checked = (i == 0);
}

// Create eight buttons for terminating dialog box.

String[] astrButton = {"Reset", "Invert", "Translate", "Scale",...
EventHandlers aeh = { new EventHandlers(ButtonResetOnClick),
                    new EventHandlers(ButtonInvertOnClick),
                    new EventHandlers(ButtonTranslateOnClick),
                    new EventHandlers(ButtonScaleOnClick),
                    new EventHandlers(ButtonRotateOnClick),
                    new EventHandlers(ButtonRotateAtOnClick),
                    new EventHandlers(ButtonShearOnClick) };

for (int i = 0; i < 8; i++)
{
    Button btn = new Button();
    btn.Parent = this;
    btn.Text   = astrButton[i];
    btn.Location = new Point(100 + 72 * (i > 3 ? 1 : 0),
                          8 + (i % 4) * 24);
    btn.Size     = new Size(64, 14);

    if (i == 0)    // Reset button
    {
        AcceptButton = btn;
    }
    if (i == 1)    // Invert button
    {
        btnInvert = btn;
    }
    if (i < 7)     // All buttons except Cancel
    {
        btn.Click += aeh[i];
        btn.DialogResult = DialogResult.OK;
    }
    else           // Cancel button
    {
        btn.DialogResult = DialogResult.Cancel;
        CancelButton = btn;
    }
}

ClientSize = new Size(240, 106);

AutoSizeBaseSize = new Size(4, 8);

public Matrix Matrix
{
    get

return matrix;

set
{
    matrix = value;
    btnInvert.Enabled = matrix.IsInvertible;
}

void ButtonResetOnClick(object obj, EventArgs ea)
{
    matrix.Reset();
}
void ButtonInvertOnClick(object obj, EventArgs ea)
{
    matrix.Invert();
}
void ButtonTranslateOnClick(object obj, EventArgs ea)
{
    matrix.Translate((float) updown[0].Value,
                    (float) updown[1].Value,
                    radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
}
void ButtonScaleOnClick(object obj, EventArgs ea)
{
    matrix.Scale((float) updown[0].Value,
                  (float) updown[1].Value,
                  radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
}
void ButtonRotateOnClick(object obj, EventArgs ea)
{
    matrix.Rotate((float) updown[2].Value,
                   radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
}
void ButtonRotateAtOnClick(object obj, EventArgs ea)
{
    matrix.RotateAt((float) updown[2].Value,
                      new PointF((float) updown[0].Value,
                                  (float) updown[1].Value),
                      radio[0].Checked ? MatrixOrder.Prepend : MatrixOrder.Append);
}
void ButtonShearOnClick(object obj, EventArgs ea)
Here's a view of the program with both dialog boxes displayed:

![Image of program interface with transformation settings]

The chimney of the house is a **TextureBrush** based on a **Bitmap** image that looks like bricks. I could have used a **HatchBrush** here with **HatchStyle.HorizontalBrick**. But, as you'll recall from Chapter 11, the **HatchBrush** isn't subject to transforms; the **TextureBrush** is. As you make the house larger, the bricks of the chimney also get larger. The bricks can also be sheared and rotated. You'll also notice that GDI+ uses a smoothing algorithm to avoid a giant-pixel effect.
Chapter 19: Font Fun

Overview

The TrueType and OpenType fonts available to Windows Forms programs are outline fonts, which means that each character is defined by a series of straight lines and splines. As we saw in Chapter 9 and subsequent chapters, outline fonts are continuously scalable. The font definitions also contain built-in hints that help avoid distortions that result when scaled floating-point coordinates must be rounded to a particular pixel grid.

Because font characters are defined by a series of straight lines and curves, they integrate well with the rest of the Windows Forms graphics system. Font characters are subject to transforms, they can be colored with any brush, and the character outlines can become part of a graphics path.

With the use of these various graphics programming techniques, singly and in combination, text can transcend its customary role—that of being read—and aspire to become something like art.

Getting Started

Since Chapter 5, I've been deriving from a class named PrintableForm (implemented in PrintableForm.cs) whenever I've wanted to demonstrate some graphics technique on both the video display and the printer. A class derived from PrintableForm overrides the DoPage method to draw its graphics. DoPage is called during the form's OnPaint method to paint the client area. When you click the client area, the DoPage method is also called to display graphics on the printer.

For this chapter, I want to define a new class, named FontMenuForm, that derives from PrintableForm and also includes a single menu item labeled "Font!". The item invokes a standard Font dialog box that lets you change a field named font. The FontMenuForm class also contains a couple methods that are handy for displaying text.

FontMenuForm.cs

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class FontMenuForm : PrintableForm
{
    protected string strText = "Sample Text";
    protected Font font = new Font("Times New Roman", 24, FontStyle.Italic);

    public new static void Main()
    {
        Application.Run(new FontMenuForm());
    }

    public FontMenuForm()
    {
        Text = "Font Menu Form";
        Menu = new MainMenu();
        Menu.MenuItems.Add("&Font!", new EventHandler(MenuFontOnClick));
    }
}
```
void MenuFontOnClick(object obj, EventArgs ea)
{
    FontDialog dlg = new FontDialog();
    dlg.Font = font;

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        font = dlg.Font;
        Invalidate();
    }
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    SizeF sizef = grfx.MeasureString(strText, font);
    Brush brush = new SolidBrush(clr);

    grfx.DrawString(strText, font, brush, (cx - sizef.Width) / 2,
                    (cy - sizef.Height) / 2);
}

public float GetAscent(Graphics grfx, Font font)
{
    return font.GetHeight(grfx) *
        font.FontFamily.GetCellAscent(font.Style) /
        font.FontFamily.GetLineSpacing(font.Style);
}

public float GetDescent(Graphics grfx, Font font)
{
    return font.GetHeight(grfx) *
        font.FontFamily.GetCellDescent(font.Style) /
        font.FontFamily.GetLineSpacing(font.Style);
}

public float PointsToPageUnits(Graphics grfx, Font font)
{
    float fFontSize;

    if (grfx.PageUnit == GraphicsUnit.Display)
        fFontSize = 100 * font.SizeInPoints / 72;
    else
        fFontSize = grfx.DpiX * font.SizeInPoints / 72;

    return fFontSize;
}
A program that derives from the `FontMenuForm` class should override the `DoPage` method (as when deriving from the `PrintableForm` class) and also make use of the `font` and `strText` fields. Optionally, the program can set those two fields in its constructor to different initial values.

The `GetAscent` and `GetDescent` methods calculate the ascent and descent of a particular font by using a technique I discussed in Chapter 9. The `PointToPageUnits` method calculates the point size of a font in page units. The method assumes that the default page units are in effect for the `Graphics` object argument. Printers have default page units of `GraphicsUnit.Display`, and the video display has default page units of `GraphicsUnit.Pixel`.

**Brushed Text**

Throughout Chapter 17, I resisted demonstrating how you can use the whole variety of available brushes with text because I was saving them for this chapter. Here, for example, is a program that displays text using a `HatchBrush` created with `HatchStyle.HorizontalBrick`.

```csharp
// Bricks.cs
// Bricks.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Bricks : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new Bricks());
    }

    public Bricks()
    {
        Text = "Bricks";
        strText = "Bricks";
        font = new Font("Times New Roman", 144);
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        SizeF sizef = grfx.MeasureString(strText, font);
        Brush hbrush = new HatchBrush(HatchStyle.HorizontalBrick,
                                        Color.White, Color.Black);

        grfx.DrawString(strText, font, hbrush,
                         (cx - sizef.Width) / 2,
                         (cy - sizef.Height) / 2);
    }
}
```

When you enlarge the client area sufficiently, the output looks like this:
Hatch brushes work best with larger font styles. With some of the skimpier hatch brushes, the 
appearance can be improved by outlining the characters, a technique I'll demonstrate later in the 
chapter.

Here's a program that displays text with a gradient brush.

\texttt{GradientText.cs}

```
//-------------------------------------------
// GradientText.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class GradientText: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new GradientText());
    }
    public GradientText()
    {
        Text = "Gradient Text";
        Width *= 3;
        strText = "Gradient";
        font = new Font("Times New Roman", 144, FontStyle.Italic);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        SizeF  sizeof  = grfx.MeasureString(strText, font);
        PointF ptf = new PointF((cx - sizeof.Width) / 2,
                                (cy - sizeof.Height) / 2);

        RectangleF rectf = new RectangleF(ptf, sizeof);

        LinearGradientBrush lgbrush = new LinearGradientBrush(rectf,
                                                   Color.White, Color.Black,
                                                   GradientMode.Linear, 
                                                   LinearGradientMode.Vertical);

        grfx.FillRectangle(lgbrush, rectf);
    }
```
LinearGradientMode.ForwardDiagonal);
        grfx.Clear(Color.Gray);
        grfx.DrawString(strText, font, lgbrush, ptf);
    }
}

The text is white at the upper left corner and black at the lower right corner, displayed against a gray background:

![Gradient](image)

If you insert the lines
[sizef.Width /= 8;
sizef.Height /= 8;
]

before the `RectangleF` creation and the line
lgbrush.WrapMode = WrapMode.TileFlipXY;

after the brush creation, you get a smaller tiled brush that looks like this:

![Gradient](image)

It's also possible to achieve some interesting effects using plain old solid brushes, as illustrated in the DropShadow program.

**DropShadow.cs**

```csharp
//--------------------------------------------------
// DropShadow.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DropShadow: FontMenuForm
{
    const int iOffset = 10;  // Approximately 1/10 inch (exactly on
printer)

    public new static void Main()
```
This program does a drop-shadow effect by displaying the same text with two different brushes offset by 10 units:

If the offset is very small and you choose the colors right, you can achieve an embossed or an engraved effect, as shown in the EmbossedText program.

```
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
EmbossedText.cs
//------------------------------------------------------------------------------
// EmbossedText.cs © 2001 by Charles Petzold
//------------------------------------------------------------------------------
```
class EmbossedText: FontMenuForm
{
    int iOffset = 2;

    public new static void Main()
    {
        Application.Run(new EmbossedText());
    }

    public EmbossedText()
    {
        Text = "Embossed Text";
        Width *= 2;
        Menu.MenuItems.Add("&Toggle!",
            new EventHandler(MenuToggleOnClick));
        strText = "Emboss";
        font = new Font("Times New Roman", 108);
    }

    void MenuToggleOnClick(object obj, EventArgs ea)
    {
        iOffset = -iOffset;
        Text = (iOffset > 0) ? "Embossed Text" : "Engraved Text";
        strText = (iOffset > 0) ? "Emboss" : "Engrave";
        Invalidate();
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        SizeF sizef = grfx.MeasureString(strText, font);
        float x     = (cx - sizef.Width) / 2;
        float y     = (cy - sizef.Height) / 2;

        grfx.Clear(Color.White);
        grfx.DrawString(strText, font, Brushes.Gray, x, y);
        grfx.DrawString(strText, font, Brushes.White, x - iOffset,
            y - iOffset);
    }
}

The program draws gray text and then white text against a white background. By default, the program comes up like this:
The program includes a menu option labeled "Toggle!" that lets you switch to the engraved effect:

These two effects are fundamentally the same. The only difference is the choice of a positive or negative offset between the two text displays. Because we are accustomed to light sources that come from above, we interpret an apparent shadow that appears on the bottom and right of the characters to be the result of raised text, and a shadow on the top and left to result from sunken text. Turn this book (or your monitor) upside down to swap the effects.

As demonstrated in the BlockFont program, you can draw the same text string multiple times with the same color to achieve a block effect.

```
BlockFont.cs
//----------------------------------------
// BlockFont.cs © 2001 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BlockFont: FontMenuForm
{
    const int iReps = 50; // Approximately 1/2 inch (exactly on printer)

    public new static void Main()
    {
        Application.Run(new BlockFont());
    }
    public BlockFont()
    {
        Text = "Block Font";
        Width *= 2;
    }
```

strText = "Block";
font = new Font("Times New Roman", 108);
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    SizeF sizef = grfx.MeasureString(strText, font);
    float x     = (cx - sizef.Width  - iReps) / 2;
    float y     = (cy - sizef.Height + iReps) / 2;

    grfx.Clear(Color.LightGray);
    for (int i = 0; i < iReps; i++)
        grfx.DrawString(strText, font, Brushes.Black, x + i, y - i);
    grfx.DrawString(strText, font, Brushes.White, x + iReps,
        y - iReps);
}

After drawing multiple black text strings, the DoPage method finishes with a white one. Here's the result:

You might also want to use an outlined font on top, which will give a stronger look against a white background.

Font Transforms

It became clear from the first investigations into the world transform in Chapter 7 that text is subject to the same scaling, rotation, and shearing effects as any other graphics object. The RotatedFont program derives from FontMenuForm to draw a series of identical text strings circling the center of the display area.

RotatedFont.cs

//------------------------------
// RotatedFont.cs © 2001 by Charles Petzold
//------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class RotatedFont: FontMenuForm
{
    const int iDegrees = 20;  // Should be divisor of 360

    public new static void Main()
    {
        Application.Run(new RotatedFont());
    }

    public RotatedFont()
    {
        Text = "Rotated Font";

        strText = "   Rotated Font";
        font = new Font("Arial", 18);
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        StringFormat strfmt = new StringFormat();
        strfmt.LineAlignment = StringAlignment.Center;

        grfx.TranslateTransform(cx / 2, cy / 2);

        for (int i = 0; i < 360; i += iDegrees)
        {
            grfx.DrawString(strText, font, brush, 0, 0, strfmt);
            grfx.RotateTransform(iDegrees);
        }
    }
}

The DoPage method calls TranslateTransform to set the origin in the middle of the display area. It then draws 18 versions of the text string, each rotated an additional 20 degrees around the origin. The DrawString call uses a StringFormat object that vertically centers the text string with respect to the origin, and the text string begins with three blank characters so that there won't be a mess in the center. Here's the result with the default 18-point Arial font:
Feel free to use the program's Font! menu item to change the font to something other than Arial.

Here's a program that uses the GetAscent method in FontMenuForm. In Chapter 7, I demonstrated that negative scaling factors cause graphics objects to be flipped around either the horizontal or the vertical axis, or both. The ReflectedText program displays the text string "Reflect" using four combinations of positive and negative scaling.

ReflectedText.cs

//--------------------------------------------
// ReflectedText.cs © 2001 by Charles Petzold
//--------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ReflectedText: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new ReflectedText());
    }

    public ReflectedText()
    {
        Text = "Reflected Text";
        Width *= 2;
        strText = "Reflect";
        font = new Font("Times New Roman", 54);
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush         = new SolidBrush(clr);
        float fAscent       = GetAscent(grfx, font);
StringFormat strfmt = StringFormat.GenericTypographic;

ggfx.TranslateTransform(cx / 2, cy / 2);

for (int i = 0; i < 4; i++)
{
    GraphicsState grfxstate = grfx.Save();

    grfx.ScaleTransform((i > 1 ? -1 : 1), (i & 1) == 1 ? -1 : 1);
    grfx.DrawString(strText, font, brush, 0, -fAscent, strfmt);
    grfx.Restore(grfxstate);
}
}

The TranslateTransform call sets the origin in the center of the client area. The ScaleTransform call looks a bit messy, but it basically uses the variable i to select four different combinations of 1 and −1 for scaling. The −fAscent argument in DrawString positions the text with the left end of its baseline at the origin:

You can also combine effects. Here’s the same program with a RotateTransform call between TranslateTransform and ScaleTransform.

RotateAndReflect.cs

// RotateAndReflect.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RotateAndReflect: FontMenuForm
{
    public new static void Main()
    {

Application.Run(new RotateAndReflect());
}

public RotateAndReflect()
{
    Text = "Rotated and Reflected Text";

    strText = "Reflect";
    font = new Font("Times New Roman", 36);
}

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Brush brush = new SolidBrush(clr);
    float fAscent = GetAscent(grfx, font);
    StringFormat strfmt = StringFormat.GenericTypographic;
    grfx.TranslateTransform(cx / 2, cy / 2);

    for (int i = 0; i < 4; i++)
    {
        GraphicsState grfxstate = grfx.Save();

        grfx.RotateTransform(-45);
        grfx.ScaleTransform((i > 1 ? -1 : 1), (i & 1) == 1 ? -1 : 1);
        grfx.DrawString(strText, font, brush, 0, -fAscent, strfmt);
        grfx.Restore(grfxstate);
    }
}

I've also made the font a little smaller so that the screen shot isn't inordinately huge:
Rotation doesn't distort the individual characters. If you tilt this book side to side, you'll see that each text string displayed so far is entirely normal except that it's oriented in a different direction. When you shear a text string, however, the individual characters are distorted. The horizontal and vertical strokes will no longer be at right angles to each other.

Here's a program that uses the Shear method of the Matrix class to set a horizontal shear of 0.5.

**SimpleShear.cs**

```csharp
//------------------------------------------
// SimpleShear.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class SimpleShear : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new SimpleShear());
    }
    public SimpleShear()
    {
        Text = "Simple Shear";
        strText = "Shear";
        font = new Font("Times New Roman", 72);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
```
The particular call to `Shear` in this program changes the default transformation matrix from

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]

to

\[
\begin{pmatrix}
1 & 0 & 0 \\
0.5 & 1 & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]

and the transformation formulas from

\[
x' = x \\
y' = y
\]

to

\[
x' = x + 0.5 \cdot y \\
y' = y
\]

At the very top of the client area (or printer page), \(y\) equals 0 and the shear has no effect. But as you move down the client area, \(y\) gets larger and hence \(x'\) is shifted more to the right. Here's the resultant text string:
Although the characters bend to the left, notice that they still sit on a horizontal baseline; that is, the characters are not simply rotated but definitely distorted. This is a reverse oblique (or italic) effect. As you increase the first argument to `Shear`, the effect becomes more pronounced; if you make the argument negative, you'll get more normal-looking oblique text.

If you switch around the coordinates to the `Shear` method like so,

```csharp
matx.Shear(0, 0.5f);
```

you'll get vertical shear:

![Vertical Shear](image1)

Here you can see that the vertical strokes on the `h` and `r` are still vertical, but the baseline is now at an angle to the horizontal axis.

Using shear with text can be a bit tricky. For example, suppose you want to draw some text on a baseline, as does the TextOnBaseline program in Chapter 9 (on page 391):

![Baseline](image2)

Now suppose you want to give the text an artificial oblique look using shear. You probably want the text to sit on the same baseline in the same location, but just tilted forward a bit. Because shear is always relative to the origin, you need to use translation to move the origin to the baseline.

The BaselineTilt program is much like TextOnBaseline but has been modified to derive from FontMenuForm. The BaselineTilt program is also a little different in that it sets the baseline three-quarters of the distance down the client area rather than one-half (to take better advantage of client area real estate) and makes use of the `GetAscent` method in FontMenuForm.

**BaselineTilt.cs**

```csharp
//-------------------------------------------
// BaselineTilt.cs © 2001 by Charles Petzold
//-------------------------------------------
```
```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class BaselineTilt : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new BaselineTilt());
    }
    public BaselineTilt()
    {
        Text = "Baseline Tilt";
        strText = "Baseline";
        font = new Font("Times New Roman", 144);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        float yBaseline = 3 * cy / 4;
        float cyAscent = GetAscent(grfx, font);

        grfx.DrawLine(new Pen(clr), 0, yBaseline, cx, yBaseline);
        grfx.TranslateTransform(0, yBaseline);
        Matrix matx = grfx.Transform;
        matx.Shear(-0.5f, 0);
        grfx.Transform = matx;

        grfx.DrawString(strText, font, new SolidBrush(clr), 0, -cyAscent);
    }
}
```

This new program also shears the text string. The `TranslateTransform` call sets the origin at the designated baseline on the left side of the client area:

```
grfx.TranslateTransform(0, yBaseline);
```

The following three calls set a negative horizontal shear:

```
Matrix matx = grfx.Transform;
matx.Shear(-0.5f, 0);
grfx.Transform = matx;
```

However, because of the previous `TranslateTransform` call, the shear is relative to the new origin. Here are the transformation formulas:
\[ x' = x - 0.5 \cdot y \\
y' = y + y_{Baseline} \]

The point \((0, 0)\) in world coordinates maps to the point \((0, y_{Baseline})\) in client area coordinates, which is the designated baseline. Points above the baseline (world coordinates with negative \(y\) values) are sheared to the right. Points below the baseline (positive \(y\) values) are sheared to the left.

The program displays the text at the world coordinate \((0, -cy_{Ascent})\), where \(cy_{Ascent}\) is the height of the characters above the baseline. I'm using a \texttt{DrawString} call with no \texttt{StringFormat} argument, so the specified world coordinates indicate the point that corresponds to the upper left corner of the text string. The transformation formulas map the coordinate \((0, -cy_{Client})\) to the point \((0.5 \times cy_{Ascent}, y_{Baseline} - cy_{Ascent})\). The left side of the baseline of the text string is displayed at the point \((0, 0)\) in world coordinates, which (as I mentioned) is mapped to the point \((0, y_{Baseline})\), exactly where we want it:

![Image of Baseline text](image)

Although that certainly looks like an italic text string, it's not: the lowercase \(a\) is roman style, not italic.

Now that we know how to display normal text and sheared text on the same baseline, a very interesting technique suddenly becomes available. Here's a program that demonstrates the technique, exaggerating the shear and making the sheared text taller as well.

\texttt{TiltedShadow.cs}

```csharp
// TiltedShadow.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TiltedShadow : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new TiltedShadow());
    }
    public TiltedShadow()
    {
        Text = "Tilted Shadow";
        strText = "Shadow";
        font = new Font("Times New Roman", 54);
    }
}
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    float fAscent = GetAscent(grfx, font);

    // Set baseline 3/4 down client area.
    grfx.TranslateTransform(0, 3 * cy / 4);
    // Save the graphics state.
    GraphicsState grfxstate = grfx.Save();

    // Set scaling and shear, and draw shadow.
    grfx.MultiplyTransform(new Matrix(1, 0, -3, 3, 0, 0));
    grfx.DrawString(strText, font, Brushes.DarkGray, 0, -fAscent);

    // Draw text without scaling or shear.
    grfx.Restore(grfxstate);
    grfx.DrawString(strText, font, Brushes.Black, 0, -fAscent);
}

Like the BaselineTilt program, the TiltedShadow program calls TranslateTransform to set the origin at the client-area coordinate \((0, 3 \cdot cy / 4)\). The MultiplyTransform call then multiplies the transform by the matrix:

\[
\begin{bmatrix}
1 & 0 & 0 \\
-3 & 3 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

The resultant composite transform is

\[
x' = x - 3 \cdot y \\
y' = 3 \cdot y + (3 \cdot cy / 4)
\]

The shear is more extreme than in the earlier program, and the \(y\) coordinates are tripled as well. Combined with text displayed normally on the baseline, the result is a shadow effect:
This effect doesn't work well when some of the text characters have descenders: the shadow then appears in front of the text. To make the shadow seem to fall back from the bottom of the descenders, change the variable I've called \texttt{fAscent} to be the sum of the ascent and the descent:

\begin{verbatim}
float fAscent = GetAscent(grfx, font) + GetDescent(grfx, font);
\end{verbatim}

Of course, you'll want to change the \texttt{strText} variable to use an appropriate text string:

\begin{verbatim}
notice that the shadow falls back from the bottom of the q and the y.
\end{verbatim}

\textbf{Text and Paths}

The \texttt{GraphicsPath} class includes a method named \texttt{AddString} that lets you add a text string to a path. The straight lines and curves that make up the character outlines become part of the path. As usual, however, text is a little different from other graphics objects, and adding text to a path requires some special considerations.

The first problem involves the \texttt{AddString} method itself. As you'll recall from Chapter 15, most of the \texttt{Add} methods in \texttt{GraphicsPath} are very similar to the corresponding \texttt{Draw} methods in the \texttt{Graphics} class. For example, using the \texttt{Graphics} class, you can draw a line by calling

\begin{verbatim}
grfx.DrawLine(pen, x1, y1, x2, y2);
\end{verbatim}

You can add a line to a path by calling

\begin{verbatim}
path.AddLine(x1, y1, x2, y2);
\end{verbatim}

The \texttt{AddLine} method doesn't require a \texttt{Pen} argument because the path retains only the coordinates of the lines.

In contrast, the \texttt{AddString} methods of \texttt{GraphicsPath} are considerably different from the \texttt{DrawString} methods of the \texttt{Graphics} class. Instead of specifying a font (as in \texttt{DrawString}), you specify the three basic components that go into making a font (a font family, a style, and a size) plus a destination (either a point or a rectangle) and a \texttt{StringFormat} object:

\textit{GraphicsPath AddString Methods}
AddString(string str, FontFamily ff, int iStyle, float fSize, Point pt, StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize, PointF ptf, StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize, Rectangle rect, StringFormat sf)
AddString(string str, FontFamily ff, int iStyle, float fSize, RectangleF rectf, StringFormat sf)

Notice that the third argument is defined as an int, but it's really a member of the FontStyle enumeration cast to an int.

When you create a font for drawing, you generally base the font on a specific metrical size. Very often, you specify the font size in points, but you can also use inches or millimeters. As I showed in Chapter 9, it's also possible to create a font that is not a specific metrical size by using the GraphicsUnit.Pixel or GraphicsUnit.World value in the Font constructor. The size of such a font is just a number. When you render text using that font, the size is interpreted at that time in terms of the current page coordinates of the output device.

A path doesn't retain metrical information. The path is only a collection of coordinates. For that reason, the AddString method can't be defined in terms of a Font. (And if there were an AddString method that did include a Font argument, it would undoubtedly be restricted to a Font object created with GraphicsUnit.Pixel or GraphicsUnit.World.)

Let's assume you add a text string to a path by calling AddString with a fourth argument of 72. What does that number mean? How large will the font be? It all depends on the page transform in effect when you eventually render that path by calling DrawPath or FillPath. If the page unit is GraphicsUnit.Point, the text will be rendered in the same size as a 72-point font drawn using DrawString. But if you render the path on the printer using the default GraphicsUnit.Display page units, the 72 units will be interpreted as 0.72 inch, and the text will be rendered in the same size as a 52-point font. (That's 0.72 inch times 72 points to an inch.) If you render the path on the screen using the default GraphicsUnit.Pixel page units, the size of the font will depend on the video resolution. If the video resolution is 120 dpi, for example, the 72-unit size of the font will be interpreted as 72/120 inch and will appear about the size of a 43-point font. (That's 72/120 inch times 72 points to the inch.)

It's likely that you want text in a path to be compatible with text displayed normally. Perhaps the simplest approach is to call AddString with the desired point size of the font. Before rendering the path, you simply set page units to points:

grfx.PageUnit = GraphicsUnit.Point;

Another approach (one that I'll be using in the sample programs in the remainder of this chapter) lets you use the default page units when rendering the font. But you need to calculate an fSize argument to AddString based on the desired point size of the font and the resolution of the device. (That means you can't use the same path on both the screen and the printer.) Suppose you already have a Font object named font, and you want to add text to a path based on that font. The second argument of AddString is just font.FontFamily. The third argument is font.FontStyle cast to an int. The fourth argument is calculated like so:

if (grfx.PageUnit == GraphicsUnit.Display)
    fFontSize = 100 * font.SizeInPoints / 72;
else
    fFontSize = grfx.DpiX * font.SizeInPoints / 72;
The first calculation is for the printer; the second is for the video display. If you look back at the FontMenuForm program, you'll see that I've implemented this calculation in the PointsToPageUnits method.

Regardless of what precautions you take to calculate font sizes, text displayed using DrawString and text displayed by rendering a path on which AddString was earlier called won't be identical, particularly on the video display. When you call DrawString, some adjustments are made to the text to make it more readable. If you need two chunks of text to be rendered in exactly the same size (for example, if one overlays the other), don't use a path for one and DrawString for the other. Use a path for both.

I've mentioned hints in connection with outline fonts. When you add a text string to a path, all that's saved in the path are floating-point coordinates. The hints are gone. When that path is eventually rendered, the floating-point coordinates must be converted to pixels. Some rounding will be involved, and the rounding isn't likely to be entirely consistent. For example, the widths of the two vertical strokes of the H might differ by a pixel. For large font sizes, you won't notice the problem. On high-resolution devices like the printer, you won't notice the problem either. But for normal font sizes on the video display, the appearance of the text will be intolerable.

For that reason, text added to paths looks best in larger font sizes or on high-resolution output devices like the printer. Use a path only for special text effects; avoid using paths for normal text.

Let's look at an example. Some graphics systems let you create an outline, or hollow, font that consists of just an outline with an unfilled interior. Windows Forms doesn't include such a style in the FontStyle enumeration. To display outlined font characters, however, you can use a path, as I've done in the HollowFont program.

HollowFont.cs
<!--[105x720] -->
// HollowFont.cs © 2001 by Charles Petzold
// -----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
class HollowFont: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new HollowFont());
    }
    public HollowFont()
    {
        Text = "Hollow Font";
        Width *= 2;
        strText = "Hollow";
        font = new Font("Times New Roman", 108);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);
        fFontSize = PointsToPageUnits(grfx, font);
// Get coordinates for a centered string.

SizeF sizef = grfx.MeasureString(strText, font);
PointF ptf = new PointF((cx - sizef.Width) / 2,
                        (cy - sizef.Height) / 2);

// Add text to the path.

path.AddString(strText, font.FontFamily, (int) font.Style,
                fFontSize, ptf, new StringFormat());

// Draw the path.

grfx.DrawPath(new Pen(clr), path);
}
}

Before calling AddString, the program calculates a font size—here stored in a variable named fFontSize—using the technique I described earlier. The program also calculates a PointF argument to AddString that has the effect of centering the string in the client area when the path is drawn (which happens at the end of the DoPage method).

The HollowFont program calculates the PointF destination of the text before adding the text to the path because the DrawPath method itself has no argument to indicate where the path is drawn. All the coordinates in the path are simply interpreted as world coordinates when DrawPath is called. However, the PointF argument to AddString indicates the upper left coordinate of the text string. All the coordinates of the text characters are relative to that point, and those are the coordinates stored in the path. The HollowFont program calculates this point using MeasureString with the original Font object as if it were preparing to display text using DrawString. Here's the result:

![HollowFont]

Try clicking the Font! menu item to specify a 12-point font. You'll see that the text doesn't look very readable on the video display. It's a rounding problem.

There's another approach to centering text stored in a path. This alternative approach is more generalized because it's based on the coordinates stored within the path itself. The GraphicsPath method GetBounds returns a RectangleF that is the smallest rectangle encompassing all the coordinates of the path. You can use that rectangle to center the contents of the path, perhaps most easily by calling TranslateTransform on the destination Graphics object.

Don't assume that the rectangle returned from GetBounds will have an upper left corner of (0, 0), even if you call AddString with a PointF argument of (0, 0). The Left property of the rectangle will probably be a bit greater than 0 because there's normally a little margin before the first text character. The Top property of the rectangle will likewise often be larger than 0.
Here's a program that calls `AddString` with a `PointF` argument of (0, 0) and then centers the path in the client area using the bounding rectangle of the path.

**HollowFontCenteredPath.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontCenteredPath : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new HollowFontCenteredPath());
    }
    public HollowFontCenteredPath()
    {
        Text = "Hollow Font (Centered Path)";
        Width *= 2;
        strText = "Hollow";
        font = new Font("Times New Roman", 108);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);

        // Add text to the path.
        path.AddString(strText, font.FontFamily, (int) font.Style, fFontSize, new PointF(0, 0), new StringFormat());

        // Get the path bounds for centering.
        RectangleF rectfBounds = path.GetBounds();

        grfx.TranslateTransform((cx - rectfBounds.Width) / 2 - rectfBounds.Left, (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

        // Draw the path.
        Pen pen = new Pen(clr, fFontSize / 50);
    }
}
```
pen.DashStyle = DashStyle.Dot;

grfx.DrawPath(pen, path);
}
}

Notice the call to *TranslateTransform* based on the width and height of the display area, and the dimensions and upper left corner of the bounding rectangle.

I've also defined the pen a little differently, this time making the width equal to 1/50 of *fFontSize* and setting a *DashStyle* of *Dot*:

If you compare this screen shot with the one from HollowFont, you'll see that it's centered a little differently. The vertical centering of the HollowFont text is based on the height returned from *MeasureString*, which is a height associated with the font and includes descenders and diacritical marks, even if they're not present in the particular text string being measured. For example, *MeasureString* returns the same height for the strings "Å", "a", and "y".

The text displayed by HollowFontCenteredPath, however, is vertically centered based solely on the coordinates in the path. The centering of the text is more visually accurate.

Do you remember the HowdyWorldFullFit program in Chapter 9 that attempted to expand a string by using *MeasureString* and *ScaleTransform* to fit a rectangle? It didn't quite manage to fill the destination rectangle because *MeasureString* includes margins for descenders and diacritical marks. Here's a program that uses the path bounding rectangle and *ScaleTransform* to tightly fill the client area with a short text string.

**FullFit.cs**

```csharp
// FullFit.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
class FullFit: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new FullFit());
    }
    public FullFit()
    {
        Text = "Full Fit";
    }
}
```
strText = "Full Fit";
font = new Font("Times New Roman", 108);

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();

    // Add text to the path.

    path.AddString(strText, font.FontFamily, (int) font.Style,
                   100, new Point(0, 0), new StringFormat());

    // Set the world transform.

    RectangleF rectfBounds = path.GetBounds();
    PointF[] aptfDest = { new PointF(0, 0), new PointF(cx, 0),
                         new PointF(0, cy) };

    grfx.Transform = new Matrix(rectfBounds, aptfDest);

    // Fill the path.

    grfx.FillPath(new SolidBrush(clr), path);
}

The program sets a world transform using the powerful Matrix constructor that has two arguments, a RectangleF structure and an array of three PointF structures:

grfx.Transform = new Matrix(rectfBounds, aptfDest);

This constructor calculates a transform that maps three corners of the RectangleF structure to the three PointF structures. Simply set the RectangleF structure to the path bounding rectangle, and the three PointF structures to three corners of the display area, and the path fills the space.
You'll definitely want to print this one as well. You can change the font, but you'll see that the initial font size doesn't matter. The text is always scaled to the size of the client area. That's why the AddString call in this program has an arbitrary hard-coded font size value of 100.

I mentioned earlier that when you need to use a path for displaying text, you should also use it for any other text that must match in size, even if the other text doesn't require a path. Here's a program that uses the same path for two FillPath calls and one DrawPath call. The two FillPath calls could have been done with a call to DrawString except the text wouldn't have aligned correctly on the screen.

DropShadowWithPath.cs

//-------------------------------------------------
// DropShadowWithPath.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class DropShadowWithPath: FontMenuForm
{
    const int iOffset = 10; // Approximately 1/10 inch (exactly on printer)

    public new static void Main()
    {
        Application.Run(new DropShadowWithPath());
    }
    public DropShadowWithPath()
    {
        Text = "Drop Shadow with Path";
        Width *= 2;
        strText = "Shadow";
        font = new Font("Times New Roman", 108);
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();
    float fFontSize = PointsToPageUnits(grfx, font);

    // Get the coordinates for a centered string.
    SizeF sizef = grfx.MeasureString(strText, font);
    PointF ptf = new PointF((cx - sizef.Width) / 2,
                           (cy - sizef.Height) / 2);

    // Add text to the path.
    path.AddString(strText, font.FontFamily, (int) font.Style,
                   fFontSize, ptf, new StringFormat());

    // Clear, fill, translate, fill, and draw.
    grfx.Clear(Color.White);
    grfx.FillPath(Brushes.Black, path);
    path.Transform(new Matrix(1, 0, 0, 1, -10, -10));
    grfx.FillPath(Brushes.White, path);
    grfx.DrawPath(Pens.Black, path);
}

When you outline font characters, you can color them in the same color as the background. I prefer this drop-shadow effect to the earlier one:

You can use a similar technique for a block effect on a white background.

Here's a program that draws the outline of the font characters using a pen created from a hatch brush with a width that's 1/20 of the fFontSize value.

HollowFontWidePen.cs
//------------------------------------------------
// HollowFontWidePen.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontWidePen: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new HollowFontWidePen());
    }
    public HollowFontWidePen()
    {
        Text = "Hollow Font (Wide Pen)";
        Width *= 2;
        strText = "Wide Pen";
        font = new Font("Times New Roman", 108,
                            FontStyle.Bold | FontStyle.Italic);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);

        // Add text to the path.
        path.AddString(strText, font.FontFamily, (int) font.Style,
                        fFontSize, new PointF(0, 0), new StringFormat());

        // Get the path bounds for centering.
        RectangleF rectfBounds = path.GetBounds();
        grfx.TranslateTransform(
            (cx - rectfBounds.Width) / 2 - rectfBounds.Left,
            (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

        // Draw the path.
        Brush brush = new HatchBrush(HatchStyle.Trellis,
                                      Color.White, Color.Black);
        Pen pen = new Pen(brush, fFontSize / 20);
        grfx.DrawPath(pen, path);
The effect isn't quite satisfactory to my eyes:

![Wide Pen](image)

What's needed here, I think, is a border around the brush pattern. In other words, the character outlines themselves need to be outlined. Is such a thing possible? Yes, it's what happens when you call the *Widen* method of the *GraphicsPath* class. A new path is created based on the existing path as if it had been drawn with a pen of a specific width. Here's a demonstration program.

**HollowFontWidened.cs**

```csharp
// HollowFontWidened.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class HollowFontWidened: FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new HollowFontWidened());
    }
    public HollowFontWidened()
    {
        Text = "Hollow Font (Widened)";
        Width *= 2;
        strText = "Widened";
        font = new Font("Times New Roman", 108,
                      FontStyle.Bold | FontStyle.Italic);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);

        // Add text to the path.
        path.AddString(strText, font.FontFamily, (int) font.Style,
```
// Get the path bounds for centering.
RectangleF rectfBounds = path.getBounds();

grfx.TranslateTransform(
    (cx - rectfBounds.Width) / 2 - rectfBounds.Left,
    (cy - rectfBounds.Height) / 2 - rectfBounds.Top);

// Widen, fill, and draw the path.
path.Widen(new Pen(Color.Black, fFontSize / 20));
Brush brush = new HatchBrush(HatchStyle.Trellis,
    Color.White, Color.Black);
grfx.DrawPath(new Pen(Color.Black, 2), path);
grfx.FillPath(brush, path);
}
}

All the new code appears at the bottom of the DoPage method. The previous program called DrawPath using a wide pen based on a hatch brush. This new program draws basically the same thing by calling Widen based on a wide pen and then calling FillPath using the hatch brush.

You can verify that the results are the same by commenting out the DrawPath call in the HollowFontWidened program. But the DrawPath call is the one that provides the effect I was after. It outlines the outlines:

I arrived at some of the coding decisions in HollowBrushWidened through experimentation. I originally called DrawPath after FillPath, but DrawPath drew a bunch of little squiggles and loops that are a result of the algorithm used in widening the path. (Try leaving out the FillPath call to see what I'm talking about. You may actually like the effect—like a mechanical drawing machine a bit out of alignment.) I moved FillPath after DrawPath to cover up the squiggles and then needed to use a pen width of 2 in DrawPath because the brush was obscuring much of the pen.

You can also set a clipping region from a path, which means that you can clip graphics output to the interior of a text string. Here's a program that sets a clipping region and then draws a bunch of Bézier splines in random colors.

ClipText.cs
//--------------------------------------------------
// ClipText.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClipText : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new ClipText());
    }
    public ClipText()
    {
        Text = "Clip Text";
        Width *= 2;
        strText = "Clip Text";
        font = new Font("Times New Roman", 108, FontStyle.Bold);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        GraphicsPath path = new GraphicsPath();
        float fFontSize = PointsToPageUnits(grfx, font);

        // Add text to the path.
        path.AddString(strText, font.FontFamily, (int) font.Style,
                       fFontSize, new PointF(0, 0), new StringFormat());
        // Set the clipping region.
        grfx.SetClip(path);
        // Get the path bounds and center the clipping region.
        RectangleF rectfBounds = path.GetBounds();
        grfx.TranslateClip((cx - rectfBounds.Width) / 2 - rectfBounds.Left,
                           (cy - rectfBounds.Height) / 2 - rectfBounds.Top);
        // Draw clipped lines.
        Random rand = new Random();
        for (int y = 0; y < cy; y++)
        {
            Pen pen = new Pen(Color.FromArgb(rand.Next(255),
Here's the result:

![Image of text after Warp transform]

**Nonlinear Transforms**

The matrix transform is widely available throughout the Windows Forms graphics system. You can apply it to a `Graphics` object, to a path, and to brushes and pens. But the matrix transform is always a *linear* transform. Parallel lines are always mapped to other parallel lines, and after awhile, you might begin to see those consistently parallel lines as bars on a jail cell.

The `GraphicsPath` class has one nonlinear transform available through a method named `Warp`. Here's a program that stores some text in a path and then uses `Warp` to scrunch together the top of the path.

**WarpText.cs**

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WarpText : Form
{
    int iWarpMode = 0;

    public new static void Main()
    {
        Application.Run(new WarpText());
    }

    public WarpText()
    {
```

```csharp
```
Text = "Warp Text - " + (WarpMode) iWarpMode;
Menu.MenuItems.Add("&Toggle!",
    new EventHandler(MenuToggleOnClick));
strText = "WARP";
font = new Font("Arial Black", 24);
}
void MenuToggleOnClick(object obj, EventArgs ea)
{
iWarpMode ^= 1;
Text = "Warp Text - " + (WarpMode) iWarpMode;
Invalidate();
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();

    // Add text to the path.
    path.AddString(strText, font.FontFamily, (int) font.Style, 100, new PointF(0, 0), new StringFormat());

    // Warp the path.
    RectangleF rectfBounds = path.GetBounds();
    PointF[] aptfDest = { new PointF(cx / 3, 0),
        new PointF(2 * cx / 3, 0),
        new PointF(0, cy),
        new PointF(cx, cy) };

    path.Warp(aptfDest, rectfBounds, new Matrix(), (WarpMode) iWarpMode);

    // Fill the path.
    grfx.FillPath(new SolidBrush(clr), path);
}

Earlier, in the FullFit program, I used a constructor of Matrix that calculated a transform that mapped three corners of a rectangle into three points in an array. The Warp method is similar, but it maps four corners of a rectangle (again, I use the bounding rectangle of the path) into four points of an array. For the destination array, I used the two bottom corners of the client area and two points at the top of the client area closer to the center than the top corners.
The menu item labeled "Toggle!" lets you switch between perspective and bilinear warp modes.

Although the Warp method is the only nonlinear transform directly available to Windows Forms programmers, the graphics path actually gives you the ability to perform any nonlinear transform that you can conceive and describe mathematically. Here's how you do it:

1. Get the array of coordinates in the path by using the PathPoints property.
2. Modify those coordinates by using custom transformation formulas.
3. Create a new path based on the modified coordinates.

The hard part, of course, is devising the transformation formulas.

Here's a program that displays a string of text that gets progressively taller toward the center.

```csharp
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class TallInTheCenter : FontMenuForm
{
    public new static void Main()
    {
        Application.Run(new TallInTheCenter());
    }
    public TallInTheCenter()
    {
        Text = "Tall in the Center";
        Width *= 2;
        strText = Text;
        font = new Font("Times New Roman", 48);
    }
}
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();
    float fFontSize = PointsToPageUnits(grfx, font);

    // Add text to the path.
    path.AddString(strText, font.FontFamily, (int) font.Style,
                   fFontSize, new PointF(0, 0), new StringFormat());

    // Shift the origin to the center of the path.
    RectangleF rectf = path.GetBounds();
    path.Transform(new Matrix(1, 0, 0, 1,
                              -(rectf.Left + rectf.Right) / 2,
                              -(rectf.Top + rectf.Bottom) / 2));
    rectf = path.GetBounds();

    // Modify the path.
    PointF[] aptf = path.PathPoints;
    for (int i = 0; i < aptf.Length; i++)
    path = new GraphicsPath(aptf, path.PathTypes);

    // Fill the path.
    grfx.TranslateTransform(cx / 2, cy / 2);
    grfx.FillPath(new SolidBrush(clr), path);
}

One helpful technique in programs like this is to prepare the path for a nonlinear transform by first performing a linear transform. After obtaining the path bounds (stored in the rectf variable), the program shifts the origin to the center of the path:

    path.Transform(new Matrix(1, 0, 0, 1,
                              -(rectf.Left + rectf.Right) / 2,
                              -(rectf.Top + rectf.Bottom) / 2));

The program then calls GetBounds again to store the new bounds in rectf.

To begin the nonlinear transform, the program obtains the array of PointF structures that make up the path:

    PointF[] aptf = path.PathPoints;
It then modifies the points, making the Y-coordinates larger depending on how close the points are to the center:

```csharp
for (int i = 0; i < aptf.Length; i++)
```

Then it creates a new path:

```csharp
path = new GraphicsPath(aptf, path.PathTypes);
```

Here's what it looks like:

![Tall in the Center](image)

As in the FullFit program, the initial size of the font doesn't matter.

Here's another example. This one is called WrapText (not WarpText), and it wraps a text string around the circumference of a circle. The initial size of the font doesn't matter in this program either, but the program needs to know the character ascent, so it bases the AddString call on an actual font.

**WrapText.cs**

```csharp
// WrapText.cs © 2001 by Charles Petzold
// ----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WrapText: FontMenuForm
{
    float fRadius = 100;

    public new static void Main()
    {
        Application.Run(new WrapText());
    }
    public WrapText()
    {
        Text = "Wrap Text";

        strText = "e snake ate the tail of th";
```
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    GraphicsPath path = new GraphicsPath();
    float fFontSize = PointsToPageUnits(grfx, font);

    // Add text to the path.
    path.AddString(strText, font.FontFamily, (int) font.Style,
                   fFontSize, new PointF(0, 0), new StringFormat());

    // Shift the origin to left baseline, y increasing up.
    RectF rectf = path.GetBounds();
    path.Transform(new Matrix(1, 0, 0, -1, -rectf.Left,
                       GetAscent(grfx, font)));  

    // Scale so width equals 2*PI.
    float fScale = 2 * (float) Math.PI / rectf.Width;
    path.Transform(new Matrix(fScale, 0, 0, fScale, 0, 0));

    // Modify the path.
    PointF[] aptf = path.PathPoints;

    for (int i = 0; i < aptf.Length; i++)
        aptf[i] = new PointF(
            fRadius * (1 + aptf[i].Y) * (float) Math.Cos(aptf[i].X),
            fRadius * (1 + aptf[i].Y) * (float) Math.Sin(aptf[i].X));

    path = new GraphicsPath(aptf, path.PathTypes);

    // Fill the path.
    grfx.TranslateTransform(cx / 2, cy / 2);
    grfx.FillPath(new SolidBrush(clr), path);
}
In this program, initially performing matrix transforms on the path was crucial to making the transformation formulas as simple as they are. The path is given an origin at the left side of the text baseline and is scaled so that it has a width of 2\(\pi\). The \(X\) coordinates of the path are thus simply an angle in radians and can be passed directly to the \(\text{Cos}\) and \(\text{Sin}\) methods. The results of the \(\text{Cos}\) and \(\text{Sin}\) methods are scaled by the \(Y\) coordinates in combination with the desired radius of the circle:

Look closely and you'll see that the baselines of each character are arched and that each character is wider at the top than at the bottom. Use a shorter text string to exaggerate the effect.
Chapter 20: Toolbars and Status Bars

Overview

In providing a modern user interface for your programs, toolbars and status bars are the next step beyond the standard menu and dialog boxes. Although quite different in functionality—a toolbar usually contains pictorial buttons that often duplicate menu items, whereas a status bar usually conveys textual information to the user—these two types of controls have several similarities. Both the ToolBar class and the StatusBarItem class are descended from Control, and both controls are customarily docked on an edge of the client area, the toolbar traditionally at the top and the status bar at the bottom. Both controls are usually host to smaller items. A toolbar is made up of multiple ToolBarButton items, and a status bar usually has multiple panels that are instances of the StatusBarItemPanel class. Both ToolBarButton and StatusBarItemPanel are descended from Component.

As you'll recall, when you add a menu to a form, the menu doesn't lie on top of the form's client area. Instead, the client area is reduced in size to accommodate the menu. Similarly, when you enable the auto-scroll feature of your form, the client area is reduced in size to accommodate the scroll bars.

Toolbars and status bars are not like menus and scroll bars, however. They may seem architecturally similar, but toolbars and status bars are controls just like buttons and text boxes. They have a Parent property and sit on top of their parent. When the parent of a toolbar or status bar is a form, the control sits on top of the client area. The client area is not reduced in size to accommodate the control. (This difference is mostly an historical legacy. Menus and scroll bars existed from the early days of Windows and hence were considered part of the standard application window. Toolbars and status bars were introduced later and weren't integrated into the architecture of the standard window.)

The implications are simple: When you include a toolbar or status bar in your form, you can't draw over the entire height of your client area. The toolbar hides the top of your client area, and the status bar hides the bottom. The problem is also fairly simple to solve. Instead of drawing on your client area, you'll want to draw on a Panel control that fills the space between the toolbar and the status bar.

Real-world applications that implement a toolbar or a status bar should give the user an option to hide these items. The user can decide whether these items are worth the space they take away from the client area. You can implement a View menu to provide such options. Showing or hiding the toolbar or the status bar is just a matter of toggling the control's Visible property.

The status bar is the simpler of the two controls, so let's begin with that.

The Basic Status Bar

If you need a status bar to display only one item—a description of menu items, for example—you can create a status bar and give it some text with three lines of code, as the SimpleStatusBar program illustrates.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleStatusBar: Form
{
    public static void Main()
    {
        Application.Run(new SimpleStatusBar());
    }
}
```
public SimpleStatusBar()
{
    Text = "Simple Status Bar";
    ResizeRedraw = true;

    // Create status bar.

    StatusBar sb = new StatusBar();
    sb.Parent = this;
    sb.Text = "My initial status bar text";
}

protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    Pen pen = new Pen(ForeColor);

    grfx.DrawLine(pen, 0, 0, ClientSize.Width, ClientSize.Height);
    grfx.DrawLine(pen, ClientSize.Width, 0, 0, ClientSize.Height);
}

Of course, in a real program, you wouldn't store the StatusBar object as a local variable of the constructor because you need access to it while processing events. I added an OnPaint method just to drive home the fact that the status bar hides the bottom of the client area:

![Simple Status Bar](image)

Notice that the diagonal lines seem to stop before reaching the corners of the form. The lines actually go all the way to the corners, but they're covered by the status bar.

The StatusBar control doesn't need to be explicitly positioned or sized; it extends across the full width of the form at the bottom. As you resize the form, the status bar also changes size. This behavior is a result of the Dock property of StatusBar being initialized to DockStyle.Bottom by default. A sizing grip appears at the right side of the status bar. This gives the user a larger target for grabbing the lower left corner of the form to resize it. You can remove the sizing grip by setting the SizingGrip property to false:
**StatusBar Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>SizingGrip</td>
<td>get/set</td>
</tr>
</tbody>
</table>

*SizingGrip* is one of the few properties that *StatusBar* implements itself; most of the *StatusBar* properties are inherited from *Control*.

The *BackColor* and *ForeColor* properties of the status bar are initialized to *SystemColors.Control* and *SystemColors.ControlText*, and they can't be changed. (You could, however, change the form's *BackColor* property to make the status bar stand out.) The *BackgroundImage* property has no effect on the status bar. The status bar has no border style.

The *TabStop* property of the status bar is initialized to *false*, and you'll probably not want to change it. If you have other controls in your client area, you probably don't want the Tab key to give the status bar input focus.

**StatusBar Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>TabStop</td>
<td>get/set</td>
</tr>
</tbody>
</table>

If you want to display multiple items in a status bar, you need to make use of status bar *panels*. These panels have a much wider variety of options than the status bar itself. I'll be discussing status bar panels shortly.

**The Status Bar and Auto-Scroll**

I've already warned you how the status bar affects the visibility of your client area. Here's another warning: You definitely don't want to enable the auto-scroll feature of your form when using a status bar. The status bar is treated just like any other control that is docked to the bottom. It becomes part of the display area that is scrolled by the scroll bars.

Here's a program that sets its *AutoScroll* property to *true* to demonstrate the problem.

```csharp
class StatusBarAndAutoScroll : Form
{

    public static void Main()
    {
        Application.Run(new StatusBarAndAutoScroll());
    }
    public StatusBarAndAutoScroll()
    {
        Text = "Status Bar and Auto-Scroll";
        AutoScroll = true;

        // Create status bar.
    }
}
```
StatusBar sb = new StatusBar();
sb.Parent = this;
sb.Text = "My initial status bar text";

    // Create labels as children of the form.

Label label = new Label();
label.Parent = this;
label.Text = "Upper left";
label.Location = new Point(0, 0);

label = new Label();
label.Parent = this;
label.Text = "Lower right";
label.Location = new Point(250, 250);
label.AutoSize = true;
}
}

When you run this program, you need to scroll to the bottom of the client area to see the status bar! Here's what the program looks like when you scroll to the bottom right corner of the client area:

![Image of status bar and labels](image)

This doesn't look right at all. The status bar shouldn't be affected by the scroll bars. In fact, the status bar shouldn't even be within the area surrounded by the scroll bars. It should appear below the horizontal scroll bar.

To get this program looking and working right—and to solve the painting problem that results from the reduction of the client area—you can use the Panel control. A program that uses toolbars or status bars should almost always begin with the creation of a Panel control (or other control) that initially fills the client area.

You can persuade the Panel control to fill the client area by setting its Dock property to DockStyle.Fill. Then just about anything you'd normally do with the client area (decorate it with controls or paint on it) you would do to the Panel control. The only controls that remain children of the Form rather than the Panel are the toolbar and the status bar.
Here's an example.

SimpleStatusBarWithPanel.cs

//-------------------------------------------------------
// SimpleStatusBarWithPanel.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleStatusBarWithPanel: Form
{
    public static void Main()
    {
        Application.Run(new SimpleStatusBarWithPanel());
    }
    public SimpleStatusBarWithPanel()
    {
        Text = "Simple Status Bar with Panel";

        // Create panel.
        Panel panel = new Panel();
        panel.Parent = this;
        panel.BackColor = SystemColors.Window;
        panel.ForeColor = SystemColors.WindowText;
        panel.AutoScroll = true;
        panel.Dock = DockStyle.Fill;
        panel.BorderStyle = BorderStyle.Fixed3D;

        // Create status bar as child of form.
        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.Text = "My initial status bar text";

        // Create labels as children of panel.
        Label label = new Label();
        label.Parent = panel;
        label.Text = "Upper left";
        label.Location = new Point(0, 0);
        label = new Label();
        label.Parent = panel;
        label.Text = "Lower right";
Notice that this program sets the `AutoSize` property of the `Panel` object to `true` and creates two `Label` objects as children of the panel.

The `Panel` is given a `Dock` property of `DockStyle.Fill` so that it initially fills the client area. When the `StatusBar` is then created, it assigns itself a `Dock` property of `DockStyle.Bottom`. The `Panel` and the `StatusBar` then essentially divide the client area into two nonoverlapping parts. With the panel's `AutoScroll` property set to `true`, the status bar appears below the scroll bars:

![Status Bar Panels](image)

**Status Bar Panels**

In the examples shown so far, the status bar is barely more functional than a label control that's docked to the bottom of the form. If you hide the sizing grip, you'd be hard pressed to tell them apart.

The status bar becomes more versatile when you make use of `StatusBarPanel` objects. A status bar can contain zero or more status bar panels. `StatusBar` has two properties that involve the `StatusBarPanel` objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td><code>ShowPanels</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>StatusBar.StatusBarPanelCollection</code></td>
<td><code>Panels</code></td>
<td>get</td>
</tr>
</tbody>
</table>

The `ShowPanels` property is initially `false`, and any panels that the status bar contains are ignored. Only one text string is displayed, which is the `Text` property of the `StatusBar` object itself. When you set the `ShowPanels` property to `true`, the `Text` property of `StatusBar` is ignored, and instead the text strings associated with the `StatusBarPanel` objects are displayed. Each status bar panel is associated with a text string and an optional icon. Or you can use the owner-draw facility of the `StatusBar` class.

The `Panels` property of `StatusBar` is yet another collection class that provides an indexer and implements methods such as `Add`, `AddRange`, `Insert`, and `Remove` to maintain a collection of `StatusBarPanel` objects.

Here's a simple program that creates a status bar with two panels.
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoStatusBarPanels : Form
{
    public static void Main()
    {
        Application.Run(new TwoStatusBarPanels());
    }
    public TwoStatusBarPanels()
    {
        Text = "Two Status Bar Panels";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.ShowPanels = true;

        StatusBarPanel sbpanel1 = new StatusBarPanel();
        sbpanel1.Text = "Panel 1";

        StatusBarPanel sbpanel2 = new StatusBarPanel();
        sbpanel2.Text = "Panel 2";

        sb.Panels.Add(sbpanel1);
        sb.Panels.Add(sbpanel2);
    }
}

This program creates one StatusBar object and two StatusBarPanel objects. Here's what it looks like:
StatusBarPanel doesn't derive from Control. The class has no BackColor, ForeColor, or Font property, no Location or Size property. (There is, however, a Width property, and that's initialized to 100 pixels.) StatusBarPanel has a Parent property, but it's read-only.

The only way to associate a StatusBarPanel object with a StatusBar object is through the Panels property of StatusBar. The TwoStatusBarPanels program shows the most common means of making this association, which is to use the Add method of StatusBar.StatusBarPanelCollection. Another approach is to make an array of StatusBarPanel objects and use the AddRange method:

```csharp
sb.Panels.AddRange(new StatusBarPanel[] { sbpanel1, sbpanel2 });
```

It's also possible to skip the explicit creation of the StatusBarPanel objects and use an overload of the Add method that requires just a string:

```csharp
sb.Panels.Add("Panel 1");
sb.Panels.Add("Panel 2");
```

If you need to get access to the particular StatusBarPanel to set the Text property (for example), you can simply index the Panels property like an array:

```csharp
sb.Panels[1].Text = "New panel 2 text";
```

### StatusBarPanel Properties

StatusBarPanel has 10 noninherited properties. The only read-only property indicates the StatusBar control that the particular StatusBarPanel is associated with:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusBar</td>
<td>Parent</td>
<td>get</td>
</tr>
</tbody>
</table>

The following properties of StatusBarPanel involve the panel's display of text:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusBarPanelStyle</td>
<td>Style</td>
<td>get/set</td>
</tr>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
<tr>
<td>Icon</td>
<td>Icon</td>
<td>get/set</td>
</tr>
<tr>
<td>string</td>
<td>ToolTipText</td>
<td>get/set</td>
</tr>
</tbody>
</table>
StatusBarPanelStyle is an enumeration that basically indicates whether the panel should be flagged as owner-draw:

<table>
<thead>
<tr>
<th>StatusBarPanelStyle Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>Text</td>
</tr>
<tr>
<td>OwnerDraw</td>
</tr>
</tbody>
</table>

StatusBarPanel doesn't implement any events of its own. The StatusBar class has a DrawItem event, and the event arguments contain the StatusBarPanel object that should be drawn.

The Text property is undoubtedly the most commonly used property of StatusBarPanel. The panel can also contain an icon that appears left of the text. You'll want to experiment with sizing the icon so that it fits correctly in the status bar.

The ToolTipText property is handy. Set it to a short descriptive text string that's displayed when the mouse cursor hovers over the panel. I'll demonstrate the ToolTipText property in the next program, DateAndTimeStatus.

The following properties affect the size and appearance of each panel:

<table>
<thead>
<tr>
<th>StatusBarPanel Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>StatusBarPanelBorderStyle</td>
</tr>
<tr>
<td>HorizontalAlignment</td>
</tr>
<tr>
<td>StatusBarPanelAutoSize</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

Each panel can have a different border style based on members of this enumeration:

<table>
<thead>
<tr>
<th>StatusBarPanelBorderStyle Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Raised</td>
</tr>
<tr>
<td>Sunken</td>
</tr>
</tbody>
</table>

The default is Sunken.

The Alignment property affects text within the panel, not the orientation of the panel itself. It uses the HorizontalAlignment enumeration that other controls also use:

<table>
<thead>
<tr>
<th>HorizontalAlignment Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td>Left</td>
</tr>
<tr>
<td>Right</td>
</tr>
<tr>
<td>Center</td>
</tr>
</tbody>
</table>

By default, the size of each panel is based on its Width property, which by default is 100 pixels. You can use the AutoSize property to size panels based on their text contents or to use the space left over:

<table>
<thead>
<tr>
<th>StatusBarPanelAutoSize Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Member</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Spring</td>
</tr>
<tr>
<td>Contents</td>
</tr>
</tbody>
</table>

The `MinWidth` property affects only status bar panels with an `AutoSize` setting of `StatusBarPanelAutoSize.Spring`. The default is 10 pixels.

One conventional strategy is to make the first status bar panel (the leftmost one) have a border style of `StatusBarPanelBorderStyle.None` with an `AutoSize` setting of `StatusBarPanelAutoSize.Spring`. You’d probably use this panel for menu help strings. The remaining panels have a border style of `StatusBarPanelBorderStyle.Sunken` with an `AutoSize` property of `StatusBarPanelAutoSize.None` or `StatusBarPanelAutoSize.Contents`.

Here’s a program that creates a status bar with three panels just as I described. The two panels at the right of the status bar display the current date and time.

```cs
using System;
using System.Drawing;
using System.Windows.Forms;

class DateAndTimeStatus : Form
{
    StatusBarPanel sbpMenu, sbpDate, sbpTime;

    public static void Main()
    {
        Application.Run(new DateAndTimeStatus());
    }

    public DateAndTimeStatus()
    {
        Text = "Date and Time Status";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        // Create status bar.
        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.ShowPanels = true;

        // Create status bar panels.
        sbpMenu = new StatusBarPanel();
        sbpMenu.Text = "Reserved for menu help";
    }
}
```
sbpMenu.BorderStyle = StatusBarPanelBorderStyle.None;
sbpMenu.AutoSize = StatusBarPanelAutoSize.Spring;

sbpDate = new StatusBarPanel();
sbpDate.AutoSize = StatusBarPanelAutoSize.Contents;
sbpDate.ToolTipText = "The current date";

sbpTime = new StatusBarPanel();
sbpTime.AutoSize = StatusBarPanelAutoSize.Contents;
sbpTime.ToolTipText = "The current time";

    // Attach status bar panels to status bar.
    sb.Panels.AddRange(new StatusBarPanel[]
            { sbpMenu, sbpDate, sbpTime });

    // Set the timer for 1 second.
    Timer timer = new Timer();
timer.Tick += new EventHandler(TimerOnTick);
timer.Interval = 1000;
timer.Start();
}

void TimerOnTick(object obj, EventArgs ea)
{
    DateTime dt = DateTime.Now;

    sbpDate.Text = dt.ToShortDateString();
    sbpTime.Text = dt.ToShortTimeString();
}

This program also adds ToolTipText properties to two of the status bar panels. Here's what the status bar looks like:
One of the primary uses of a status bar is to provide menu help, which involves the display of short text strings that describe each menu item as the user moves the mouse or cursor over the items.

Providing menu help takes advantage of a couple features of menus and forms that don't show up in more conventional uses of the menu. As you may recall, the MenuItem class implements five events. So far in this book, I've made use of the Popup event when a pop-up menu is first displayed and the Click event when the user picks an item from the menu. I've also demonstrated how to use the MeasureItem and DrawItem events for implementing owner-draw menu items.

The fifth MenuItem event is named Select, and it occurs whenever the mouse cursor passes over a menu item or the user presses the arrow keys to move among menu items. Select is the event you must handle to display menu help. As the cursor moves among menu items, you display a text string in the status bar for each item.

When you first try implementing menu help in a Select event handler, you'll encounter an annoying problem: the last menu help string remains in the status bar after the user finally clicks a menu item! At that point, you might seek a solution by resetting the status bar text during every Click event handler installed for the MenuItem objects. But then what happens when the user dismisses a menu by pressing the Esc key? Again, the last menu help string remains in the status bar.

There is a solution to this problem. To provide you with a way to initialize and clean up a session of menu help, the Form class has the following two events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuStart</td>
<td>OnMenuStart</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>MenuComplete</td>
<td>OnMenuComplete</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

These events signal the beginning and end of a menu operation. The crucial one is MenuComplete. When that event occurs, you want to reset the menu help text. Perhaps you want to set it to the text string that was displayed in the status bar at the time you received a MenuStart event.

That's the strategy the following program uses. It creates a status bar with a single status bar panel and uses that panel to display the menu help text. The status bar panel text is initialized to the string "Ready". The OnMenuStart method saves that text, and the OnMenuComplete method restores it.

The program installs the same Select event handler for all the menu items. That handler is the method MenuOnSelect. But because MenuHelpFirstTry is a simple demonstration program, it doesn't install any Click event handlers for the menu items.

```csharp
//-----------------------------------------------
// MenuHelpFirstTry.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuHelpFirstTry: Form
{
    StatusBarPanel sbpMenuHelp;
    string strSavePanelText;

    public static void Main()
    {
        Application.Run(new MenuHelpFirstTry());
    }

    // -----------------------------
    // MenuHelpFirstTry.cs © 2001 by Charles Petzold
    // -----------------------------
    //
    // MenuHelpFirstTry.cs © 2001 by Charles Petzold
    //
    using System;
    using System.Drawing;
    using System.Windows.Forms;

    class MenuHelpFirstTry: Form
    {
        StatusBarPanel sbpMenuHelp;
        string strSavePanelText;

        public static void Main()
        {
            Application.Run(new MenuHelpFirstTry());
        }
```
public MenuHelpFirstTry()
{
    Text = "Menu Help (First Try)";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    // Create a status bar with one panel.
    StatusBar sb = new StatusBar();
    sb.Parent = this;
    sb.ShowPanels = true;

    sbpMenuHelp = new StatusBarPanel();
    sbpMenuHelp.Text = "Ready";
    sbpMenuHelp.AutoSize = StatusBarPanelAutoSize.Spring;

    sb.Panels.Add(sbpMenuHelp);

    // Construct a simple menu.
    // For this demo, we can ignore the Click handlers,
    // but what we really need are Select handlers.
    Menu = new MainMenu();
    EventHandler ehSelect = new EventHandler(MenuOnSelect);

    // Create File menu items.
    MenuItem mi = new MenuItem("File");
    mi.Select += ehSelect;
    Menu.MenuItems.Add(mi);

    mi = new MenuItem("Open");
    mi.Select += ehSelect;
    Menu.MenuItems[0].MenuItems.Add(mi);

    mi = new MenuItem("Close");
    mi.Select += ehSelect;
    Menu.MenuItems[0].MenuItems.Add(mi);

    mi = new MenuItem("Save");
    mi.Select += ehSelect;
    Menu.MenuItems[0].MenuItems.Add(mi);

    // Create Edit menu items.
mi = new MenuItem("Edit");
mi.Select += ehSelect;
Menu.MenuItems.Add(mi);

mi = new MenuItem("Cut");
mi.Select += ehSelect;
Menu.MenuItems[1].MenuItems.Add(mi);

mi = new MenuItem("Copy");
mi.Select += ehSelect;
Menu.MenuItems[1].MenuItems.Add(mi);

mi = new MenuItem("Paste");
mi.Select += ehSelect;
Menu.MenuItems[1].MenuItems.Add(mi);
}

protected override void OnMenuStart(EventArgs ea)
{
    strSavePanelText = sbpMenuHelp.Text;
}

protected override void OnMenuComplete(EventArgs ea)
{
    sbpMenuHelp.Text = strSavePanelText;
}

void MenuOnSelect(object obj, EventArgs ea)
{
    MenuItem mi = (MenuItem) obj;
    string str;

    switch (mi.Text)
    {
    case "File":   str = "Commands for working with files";
        break;
    case "Open":   str = "Opens an existing document";
        break;
    case "Close":  str = "Closes the current document";
        break;
    case "Save":   str = "Saves the current document";
        break;
    case "Edit":   str = "Commands for editing the document";
        break;
    case "Cut":    str = "Deletes the selection and " +
        "copies it to the clipboard";
        break;
    case "Copy":   str = "Copies the selection to the " +
The `MenuOnSelect` method determines the text of the selected menu item and uses that to select a menu help string, which it then uses to set the `Text` property of the `StatusBarPanel` object. Here’s the program in action:

![Menu Help (First Try)](image)

The `Select` events are generated regardless of whether the menu item is enabled, so the program displays the same text strings even if the menu items aren't currently available. Sometimes users are confused when certain items are disabled. (I know I am.) The menu help text might clarify why an item isn’t currently available.

As is, the MenuHelpFirstTry program works fine, and you may like the idea of consolidating all the menu-help text in one place, such as the `MenuOnSelect` method. I’m not real wild about it myself, though. I’d rather bind each menu-help string with the actual menu item. Regardless, the `MenuOnSelect` method definitely needs a better means of determining which item has been selected. The `switch` and `case` construction using the `Text` property of the `MenuItem` object needs to duplicate the text strings exactly, and that can be a hassle.

One approach that I find appealing is to define a new class (named `MenuItemHelp`, for example) that subclasses `MenuItem`. `MenuItemHelp` simply adds a new property named `HelpText` that stores an additional text string. You can set the `HelpText` property when creating each object of `MenuItemHelp`:

```csharp
mi.Text = "&Open...";
mi.HelpText = "Opens an existing document";
```

The `MenuOnSelect` method then becomes much simpler:

```csharp
void MenuOnSelect(object obj, EventArgs ea)
```
You can go a step further by providing another property in `MenuItemHelp` that stores the status bar panel where the help text is to be displayed. The class itself can override the `OnSelect` methods to set the help text in the status bar panel. The MenuItemHelp.cs file defines such a class that derives from `MenuItem`.

**MenuItemHelp.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuItemHelp : MenuItem
{
    // Private fields
    StatusBarPanel sbpHelpPanel;
    string strHelpText;

    // Constructors
    public MenuItemHelp()
    {
    }
    public MenuItemHelp(string strText): base(strText)
    {
    }

    // Properties
    public StatusBarPanel HelpPanel
    {
        get { return sbpHelpPanel; }  
        set { sbpHelpPanel = value;  }
    }
    public string HelpText
    {
        get { return strHelpText; }  
        set { strHelpText = value;  }
    }

    // Method override
    protected override void OnSelect(EventArgs ea)
    {
        base.OnSelect(ea);

        if (HelpPanel != null)
            HelpPanel.Text = HelpText;
    }
}
```
Here's a revised version of the MenuHelpFirstTry program that creates `MenuItemHelp` objects rather than `MenuItem` objects. Each `MenuItemHelp` object is assigned its `HelpPanel` and `HelpText` properties as it's being created.

**MenuHelpSubclass.cs**

```csharp
// MenuHelpSubclass.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class MenuHelpSubclass: Form
{
    StatusBarPanel sbpMenuHelp;
    string strSavePanelText;

    public static void Main()
    {
        Application.Run(new MenuHelpSubclass());
    }

    public MenuHelpSubclass()
    {
        Text = "Menu Help ";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;

        // Create a status bar with one panel.
        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.ShowPanels = true;

        sbpMenuHelp = new StatusBarPanel();
        sbpMenuHelp.Text = "Ready";
        sbpMenuHelp.AutoSize = StatusBarPanelAutoSize.Spring;

        sb.Panels.Add(sbpMenuHelp);

        // Construct a simple menu with MenuItemHelp items.
        Menu = new MainMenu();

        // Create File menu items.
    }
}
```
protected override void OnMenuStart(EventArgs ea)
{
    MenuItemHelp mi = new MenuItemHelp("&File");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Commands for working with files";
    Menu.MenuItems.Add(mi);

    mi = new MenuItemHelp("&Open...");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Opens an existing document";
    Menu.MenuItems[0].MenuItems.Add(mi);

    mi = new MenuItemHelp("&Close");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Closes the current document";
    Menu.MenuItems[0].MenuItems.Add(mi);

    mi = new MenuItemHelp("&Save");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Saves the current document";
    Menu.MenuItems[0].MenuItems.Add(mi);

    // Create Edit menu items.
    mi = new MenuItemHelp("&Edit");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Commands for editing the document";
    Menu.MenuItems.Add(mi);

    mi = new MenuItemHelp("Cu&t");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Deletes the selection and " +
                   "copies it to the clipboard";
    Menu.MenuItems[1].MenuItems.Add(mi);

    mi = new MenuItemHelp("&Copy");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Copies the selection to the clipboard";
    Menu.MenuItems[1].MenuItems.Add(mi);

    mi = new MenuItemHelp("&Paste");
    mi.HelpPanel = sbpMenuHelp;
    mi.HelpText = "Replaces the current selection " +
                   "with the clipboard contents";
    Menu.MenuItems[1].MenuItems.Add(mi);
}
The Basic Toolbar

Near the end of Chapter 11, I discussed the ImageList class, which is a collection of images of the same size and color depth. A ToolBar control is basically an ImageList object and a collection of ToolBarButton objects. Each button displays one of the images in the ImageList.

Most applications use toolbar images that are 16 pixels square. That's the default size for ImageList and that's what I use throughout this chapter. If you want to use a smaller or larger button—even nonrectangular buttons—you must first create bitmaps of the desired size. (That's the obvious part.) Before adding the images to ImageList, set the ImageSize property to the size of the images. That image size will trickle through all the rest of the ToolBar and ToolBarButton objects.

These are the most essential properties of ToolBar:

<table>
<thead>
<tr>
<th>ToolBar Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>ImageList</td>
</tr>
<tr>
<td>ToolBar.ToolBarButtonCollection</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

Each ToolBar is associated with one ImageList, which, of course, usually contains multiple images.

The Buttons property is a collection of the sort we've seen several times now. You can index Buttons like an array; each element is an object of type ToolBarButton. The ToolBarButtonCollection class has several methods, including Add, AddRange, Insert, and Remove, that let you manage the buttons associated with a toolbar.

I've included the ShowToolTips property with this group of essential properties because it's fairly easy to assign ToolTips to your toolbar buttons, but you won't see them unless you set this ShowToolTips property to true.

Here are the most essential ToolBarButton properties:

<table>
<thead>
<tr>
<th>ToolBarButton Properties (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>string</td>
</tr>
</tbody>
</table>

ImageIndex is an index that indicates which image of the ImageList object is displayed on the button. ToolTipText is the text displayed when the mouse hovers over the button. The images on toolbar buttons can be fairly obscure, so ToolTips can be a big help.

I'll be talking about ToolBar events shortly, but let's first take a look at a simple do-nothing program that has a do-nothing menu and a do-nothing toolbar based on an image list based on a bitmap named StandardButtons.bmp.

SimpleToolBar.cs

//-------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleToolBar: Form
{
    public static void Main()
    {
        Application.Run(new SimpleToolBar());
    }
    public SimpleToolBar()
    {
        Text = "Simple Toolbar";

        // Create a simple menu (just for show).
        Menu = new MainMenu();
        Menu.MenuItems.Add("File");
        Menu.MenuItems.Add("Edit");
        Menu.MenuItems.Add("View");
        Menu.MenuItems.Add("Help");

        // Create ImageList.
        Bitmap bm = new Bitmap(GetType(),
                                "SimpleToolBar.StandardButtons.bmp");
        ImageList imglst = new ImageList();
        imglst.Images.AddStrip(bm);
        imglst.TransparentColor = Color.Cyan;

        // Create ToolBar.
        ToolBar tbar = new ToolBar();
        tbar.Parent = this;
        tbar.ImageList = imglst;
        tbar.ShowToolTips = true;

        // Create ToolBarButtons.
        string[] astr = {"New", "Open", "Save", "Print",
                         "Cut", "Copy", "Paste" };
    }
}
for (int i = 0; i < 7; i++)
{
    ToolBarButton tbarbtn = new ToolBarButton();
tbarbtn.ImageIndex = i;
tbarbtn.ToolTipText = astr[i];

tbar.Buttons.Add(tbarbtn);
}

StandardButtons.bmp

You can create individual bitmaps for each button, or you can create images for a bunch of buttons in one bitmap. In the SimpleToolBar program, I've used the default ImageList bitmap size of 16 pixels square. The StandardButtons.bmp file contains seven images for seven toolbar buttons, so the total size of the bitmap is 112 by 16 pixels. If you're creating a bitmap for an ImageList, the width must be an integral multiple of its height. Also, as I discussed in Chapter 11, when you include a bitmap with a project in Visual C# .NET, you must set the Build Action property to Embedded Resource.

If you don't want to experience the fun of creating your own buttons, you can use the standard buttons included with Visual Studio .NET. These are stored by default in subdirectories of the directory \Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps. The subdirectories OffCtlBr and Assorted contain bitmaps appropriate for toolbar buttons.

The SimpleToolBar program begins by creating a few top-level menu items. I added those only so that you can see how a menu and a toolbar look together. Next, the StandardButtons.bmp image is loaded as a resource. (See Chapter 11 for details on loading resources.) The program creates an ImageList object and uses the AddStrip method of the Images property to include the whole bitmap in the image list. Because the default ImageSize property indicates that the images are 16 pixels square, the ImageList object can easily figure out that there are seven images in the strip.

It's not evident from the monochrome reproduction in this book, but the background of StandardButtons.bmp is cyan, which is specified in the SimpleToolBar program as the transparent color for the ImageList.

The next step is to create the toolbar. Three properties are assigned: the Parent property indicates that the toolbar is a child of the form, the ImageList property associates the toolbar with a collection of images, and the ShowToolTips property enables ToolTips.

Next, the program creates seven ToolBarButton objects, setting the ImageIndex property to the numbers 0 through 6, corresponding to the seven images. (If the original images are not in the same order as the buttons you want to create, setting the proper ImageIndex value essentially reorders the images.) Each button also gets a short ToolTip. The program adds each button to the toolbar by using the Add method of the Buttons property:

tbar.Buttons.Add(tbarbtn);

And here's the result:
If you experiment with this program, you'll find that the buttons work much like push button controls. As you'll see later in this chapter, you can also create buttons that toggle or invoke a menu.

You'll probably want the toolbar to be distinct from the client area in some way, but you'll likely be using a `Panel` control for display purposes anyway, and you can give that a background color of `SystemColors.Window` (which is usually white).

**Toolbar Variations**

Before we get into handling events from the toolbar, let's take a look at a few properties that affect the toolbar's appearance:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td>Wrappable</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>Divider</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td><code>BorderStyle</code></td>
<td>BorderStyle</td>
<td>get/set</td>
<td><code>BorderStyle.None</code></td>
</tr>
<tr>
<td><code>ToolBarAppearance</code></td>
<td>Appearance</td>
<td>get/set</td>
<td><code>ToolBarAppearance-.Normal</code></td>
</tr>
<tr>
<td><code>ToolBarTextAlign</code></td>
<td>TextAlign</td>
<td>get/set</td>
<td><code>ToolBarTextAlign-.Underneath</code></td>
</tr>
</tbody>
</table>

When you make the form narrower than the toolbar, the `Wrappable` property indicates whether the toolbar wraps to two lines or is truncated at the right. By default, it wraps.

The `Divider` property is responsible for the line that separates the toolbar and the menu. If you set the property to `false`, the line disappears:

![Simple Toolbar](image)
You'll probably want to keep the divider for aesthetic reasons.

The `BorderStyle` property (which you can also set to `FixedSingle` or `Fixed3D`) affects the display of a border that extends the width of the form. Here's an example without a divider but with a `BorderStyle` of `FixedSingle`.

I don't think the border works as well as the divider does.

The `Appearance` property can take on one of the following properties of the `ToolBarAppearance` enumeration:

<table>
<thead>
<tr>
<th>ToolBarAppearance Enumeration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>Value</td>
</tr>
<tr>
<td>Normal</td>
<td>0</td>
</tr>
</tbody>
</table>
Applications these days seem to prefer a *Flat* appearance. Here are flat buttons with *Divider* set to *true* and no border:

![Simple Toolbar](image)

The *Flat* appearance looks better when the area underneath the toolbar is a different color.

The *TextAlign* property can take on one of the members of the *ToolBarTextAlign* enumeration:

### ToolBarTextAlign Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underneath</td>
<td>0</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
</tr>
</tbody>
</table>

This property involves a feature we haven't examined yet, which is that buttons can also include a text string:

### ToolBarButton Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
</tbody>
</table>

For example, if you include the statement

```csharp
tbarbtn.Text = astr[i];
```

in the *for* loop in SimpleToolBar, the buttons look like this:
Notice that the buttons have different sizes based on the width of the text string. If you set the TextAlign property of the toolbar to ToolBarTextAlign.Right, the buttons become wider but shorter:

**ToolBar Events**

The SimpleToolBar program doesn't handle any events, so it can't do anything in response to button clicks. In addition to the events that ToolBar inherits from Control, ToolBar implements two of its own events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonClick</td>
<td>OnButton-Click</td>
<td>ToolBarButton-ClickEventHandler</td>
<td>ToolBarButtonClickEventArgs</td>
</tr>
<tr>
<td>ButtonDropDown</td>
<td>OnButtonDown</td>
<td>ToolBarButton-ClickEventHandler</td>
<td>ToolBarButtonClickEventArgs</td>
</tr>
</tbody>
</table>

The ButtonClick event is the more important and occurs when the user clicks one of the buttons on the toolbar. The ButtonDropDown event applies only to buttons that have a certain style that is intended to invoke a menu. (I'll discuss button styles shortly.)

Notice that these two events are associated with the toolbar and not with the individual buttons. The ToolBarButton class doesn't define any events. Therefore, all button clicks on a toolbar go to a single event handler. The event handler must determine which button has been clicked. The ToolBarButtonClickEventArgs argument to the event handler has a single property that provides precisely this information:

**ToolBarButtonClickEventArgs Property**
One approach to handling the `ButtonClick` event is to first save all the `ToolBarButton` objects as fields. During the `ButtonClick` event, you then compare this `Button` property with those fields to determine how to respond to the click. Another approach is to use the `Tag` property of the `ToolBarButton` object to store something (anything) that helps you handle the event.

### `ToolBarButton` Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ToolBarButton</code></td>
<td><code>Button</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

In many cases, toolbar buttons duplicate menu items. For this reason, it makes a whole lot of sense to set the `Tag` property of the button equal to the `MenuItem` object of the menu item that the button duplicates.

Let's look at an example that's both simple and functional. The `TextBoxWithToolBar` program is a stripped-down version of the Notepad clone programs from Chapter 18. It's so stripped down that it contains only an Edit menu with Cut, Copy, and Paste items. But this program also includes a toolbar that duplicates those three items. The `ToolBar` object uses an `ImageList` based on the `StandardButtons.bmp` bitmap from the `SimpleToolBar` program. But only indices 4, 5, and 6 are used for the Cut, Copy, and Paste images.

```cs
//-------------------------------------------------
// TextBoxWithToolBar.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TextBoxWithToolBar: Form
{
    TextBox       txtbox;
    MenuItem      miEditCut, miEditCopy, miEditPaste;
    ToolBarButton tbbCut, tbbCopy, tbbPaste;

    public static void Main()
    {
        // Program doesn't run without this statement.
        Application.Run(new TextBoxWithToolBar());
    }
    public TextBoxWithToolBar()
    {
        Text = "Text Box with Toolbar";
    }
}
```

Let's look at an example that's both simple and functional. The `TextBoxWithToolBar` program is a stripped-down version of the Notepad clone programs from Chapter 18. It's so stripped down that it contains only an Edit menu with Cut, Copy, and Paste items. But this program also includes a toolbar that duplicates those three items. The `ToolBar` object uses an `ImageList` based on the `StandardButtons.bmp` bitmap from the `SimpleToolBar` program. But only indices 4, 5, and 6 are used for the Cut, Copy, and Paste images.
// Create TextBox.

    txtbox = new TextBox();
    txtbox.Parent = this;
    txtbox.Dock = DockStyle.Fill;
    txtbox.Multiline = true;
    txtbox.ScrollBars = ScrollBars.Both;
    txtbox.AcceptsTab = true;

// Create ImageList.

    Bitmap bm = new Bitmap(GetType(),
        "TextBoxWithToolBar.StandardButtons.bmp");

    ImageList imglst = new ImageList();
    imglst.Images.AddStrip(bm);
    imglst.TransparentColor = Color.Cyan;

    // Create ToolBar with ButtonClick event handler.

    ToolBar tbar = new ToolBar();
    tbar.Parent = this;
    tbar.ImageList = imglst;
    tbar.ShowToolTips = true;
    tbar.ButtonClick +=
        new
            ToolBarButtonClickEventHandler(ToolBarOnClick);

    // Create the Edit menu.

    Menu = new MainMenu();

    MenuItem mi = new MenuItem("&Edit");
    mi.Popup += new EventHandler(MenuEditOnPopup);
    Menu.MenuItems.Add(mi);

    // Create the Edit Cut menu item.

    miEditCut = new MenuItem("Cu&t");
    miEditCut.Click += new EventHandler(MenuEditCutOnClick);
    miEditCut.Shortcut = Shortcut.CtrlX;
    Menu.MenuItems[0].MenuItems.Add(miEditCut);

    // And create the Cut toolbar button.
tbbCut = new ToolBarButton();
tbbCut.ImageIndex = 4;
tbbCut.ToolTipText = "Cut";
tbbCut.Tag = miEditCut;
tbar.Buttons.Add(tbbCut);

    // Create the Edit Copy menu item.

miEditCopy = new MenuItem("&Copy");
miEditCopy.Click += new EventHandler(MenuEditCopyOnClick);
miEditCopy.Shortcut = Shortcut.CtrlC;
Menu.MenuItems[0].MenuItems.Add(miEditCopy);

    // And create the Copy toolbar button.

tbbCopy = new ToolBarButton();
tbbCopy.ImageIndex = 5;
tbbCopy.ToolTipText = "Copy";
tbbCopy.Tag = miEditCopy;
tbar.Buttons.Add(tbbCopy);

    // Create the Edit Paste menu item.

miEditPaste = new MenuItem("&Paste");
miEditPaste.Click += new EventHandler(MenuEditPasteOnClick);
miEditPaste.Shortcut = Shortcut.CtrlV;
Menu.MenuItems[0].MenuItems.Add(miEditPaste);

    // And create the Paste toolbar button.

    tbbPaste = new ToolBarButton();
tbbPaste.ImageIndex = 6;
tbbPaste.ToolTipText = "Paste";
tbbPaste.Tag = miEditPaste;
tbar.Buttons.Add(tbbPaste);

    // Set Timer for enabling buttons.

    Timer timer = new Timer();
timer.Interval = 250;
timer.Tick += new EventHandler(TimerOnTick);
timer.Start();
}

void MenuEditOnPopup(object obj, EventArgs ea)
{

miEditCut.Enabled =
miEditCopy.Enabled = (txtbox.SelectionLength > 0);
miEditPaste.Enabled =
        Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
void TimerOnTick(object obj, EventArgs ea)
{
    tbbCut.Enabled =
    tbbCopy.Enabled = (txtbox.SelectionLength) > 0;
    tbbPaste.Enabled =
        Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
{
    ToolBarButton tbb = tbbcea.Button;
    MenuItem mi = (MenuItem) tbb.Tag;

    mi.PerformClick();
}
void MenuEditCutOnClick(object obj, EventArgs ea)
{
    txtbox.Cut();
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    txtbox.Copy();
}
void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    txtbox.Paste();
}

When the constructor creates the ToolBar object, it assigns the ToolBarOnClick method as the handler for the ButtonClick event:
tbar.ButtonClick += new ToolBarButtonClickEventHandler(ToolBarOnClick);

The ToolBarButton objects are created after each corresponding MenuItem object. The Tag property of each ToolBarButton is assigned the corresponding MenuItem object. Here’s the assignment for the Cut button:
tbbCut.Tag = miEditCut;

The big payoff comes in the relative simplicity of the ToolBarOnClick method:
void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
{
    ToolBarButton tbb = tbbcea.Button;
    MenuItem mi = (MenuItem) tbb.Tag;
mi.PerformClick();
}

The method obtains the ToolBarButton object being clicked, casts the Tag property to a MenuItem, and simulates a click. This approach is so elegant that you could even reduce the body of the method to a single statement:

((MenuItem) tbbcea.Button.Tag).PerformClick();

The only messy part involves enabling and disabling the buttons based on the validity of the Cut, Copy, and Paste operations. As usual, the Cut, Copy, and Paste menu items are enabled and disabled during the Popup event for the Edit menu. However, the buttons are always in view, and they must be enabled and disabled using another technique.

If the edit control in the program were a RichTextBox rather than a TextBox, you could install an event handler for the SelectionChanged event. (TextBox doesn't define a SelectionChanged event.) Whenever the selection changes, the event handler could then enable the Cut and Copy buttons only if some text has been selected, much like the Popup event handler for the Edit menu.

That doesn't solve the problem for the Paste button, however. The Paste button must be enabled whenever there's text on the clipboard. Short of attempting to intercept Win32 messages that indicate clipboard changes, perhaps the only real solution is to create a Timer and enable the Paste button during the Tick event if text is on the clipboard. Because the Timer had to be created anyway, I decided to enable all three buttons during that Tick event.

Along with the Enabled property, the ToolBarButton class also includes a Visible property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Visible</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Enabled</td>
<td>get/set</td>
</tr>
</tbody>
</table>

**ToolBar Styles**

So far, the toolbar buttons we've created have functioned much like push buttons. You can also make a toolbar button function like a check box that can be toggled on or off. Or, a toolbar button can invoke a menu. These options are available through the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolBarButtonStyle</td>
<td>Style</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Pushed</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>PartialPush</td>
<td>get/set</td>
</tr>
<tr>
<td>Menu</td>
<td>DropDownMenu</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Despite its definition, the DropDownMenu property must actually be of type ContextMenu. Set the Style property to one of the following enumeration values:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushButton</td>
<td>1</td>
</tr>
<tr>
<td>ToggleButton</td>
<td>2</td>
</tr>
<tr>
<td>Separator</td>
<td>3</td>
</tr>
<tr>
<td>DropDownButton</td>
<td>4</td>
</tr>
</tbody>
</table>

ToolBarButtonStyle.PushButton is the default.
The `Pushed` and `PartialPush` properties apply only to buttons that have their `Style` set to `ToolBarButtonStyle.ToggleButton`. Such buttons maintain an on/off state. Each time the button is clicked, it generates a `ButtonClick` event and the value of the `Pushed` property is toggled. Your program can also initialize the state of the button by setting the `Pushed` property itself. Set the `PartialPush` property to `true` to give the button a special appearance that indicates a halfway state (much like the `ThreeState` property of the `CheckBox` control described in Chapter 12).

When you use the `ToolBarButtonStyle.Separator` style, the button ignores any image or text that may be associated with it and displays a separator instead. For toolbars that have their `Appearance` property set to `ToolBarAppearance.Flat`, the separator is a vertical line that looks much like the horizontal divider line that separates the toolbar from the menu. When the `Appearance` property is set to `ToolBarAppearance.Normal`, the separator manifests itself as a small gap between the buttons.

The `ToolBarButtonStyle.DropDownButton` option invokes a menu when the button is pressed. You specify the menu in the `DropDownMenu` property of the button. You can handle the `ButtonDropDown` event if you need to initialize the menu in some way before it's displayed.

Let's look at toggle buttons first. Here's a program that displays a four-button toolbar based on a bitmap named `FontStyleButtons.bmp`. The buttons allow you to indicate bold, italic, underline, or strikeout styles for a string of displayed text.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ToggleButtons : Form
{
    protected Panel panel;
    protected ToolBar tbar;
    protected string strText = "Toggle";
    protected Color clrText = SystemColors.WindowText;
    FontStyle fontstyle = FontStyle.Regular;

    public static void Main()
    {
        Application.Run(new ToggleButtons());
    }

    public ToggleButtons()
    {
        Text = "Toggle Buttons";

        // Create panel to fill interior.
        panel = new Panel();
        panel.Parent = this;
        panel.Dock = DockStyle.Fill;
        panel.BackColor = SystemColors.Window;
        panel/backColor = SystemColors.Window;
```
panel.ForeColor = SystemColors.WindowText;
panel.Resize += new EventHandler(PanelOnResize);
panel.Paint += new PaintEventHandler(PanelOnPaint);

    // Create ImageList.

    Bitmap bm = new Bitmap(GetType(),
    "ToggleButtons.FontStyleButtons.bmp");

    ImageList imglst = new ImageList();
    imglst.ImageSize = new Size(bm.Width / 4, bm.Height);
    imglst.Images.AddStrip(bm);
    imglst.TransparentColor = Color.White;

    // Create ToolBar.

    tbar = new ToolBar();
    tbar.ImageList = imglst;
    tbar.Parent = this;
    tbar.ShowToolTips = true;
    tbar.ButtonClick +=
        new ToolBarButtonClickEventHandler(ToolBarOnClick);

    // Create ToolBarButton.

    FontStyle[] afs = { FontStyle.Bold, FontStyle.Italic,
        FontStyle.Underline, FontStyle.Strikeout };

    for (int i = 0; i < 4; i++)
    {
        ToolBarButton tbarbtn = new ToolBarButton();
        tbarbtn.ImageIndex = i;
        tbarbtn.Style = ToolBarButtonStyle.ToggleButton;
        tbarbtn.ToolTipText = afs[i].ToString();
        tbarbtn.Tag = afs[i];

        tbar.Buttons.Add(tbarbtn);
    }

    void ToolBarOnClick(object obj, ToolBarButtonClickEventArgs tbbcea)
    {
        ToolBarButton tbarbtn = tbbcea.Button;

        // If the Tag isn't a FontStyle, don't do anything.
if (tbarbtn.Tag == null ||
    tbarbtn.Tag.GetType() != typeof(FontStyle))
return;

// Set or clear the bit in the fontstyle field.
if (tbarbtn.Pushed)
    fontstyle |= (FontStyle) tbarbtn.Tag;
else
    fontstyle &= ~(FontStyle) tbarbtn.Tag;

panel.Invalidate();
}
void PanelOnResize(object obj, EventArgs ea)
{
    Panel panel = (Panel) obj;
    panel.Invalidate();
}
void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel    panel = (Panel) obj;
    Graphics grfx  = pea.Graphics;
    Font     font  = new Font("Times New Roman", 72, fontstyle);
    SizeF    sizef = grfx.MeasureString(strText, font);

grfx.DrawString(strText, font, new SolidBrush(clrText),
                 (panel.Width - sizef.Width) / 2,
                 (panel.Height - sizef.Height) / 2);
}

FontStyleButtons.bmp

This program creates a Panel control that fills the client area (at least before the toolbar is created) and displays any output that would normally go in the client area. The PanelOnPaint event handler displays a string of centered text. Because it's displaying centered text, the panel must be repainted whenever it's resized. However, the handy ResizeRedraw property is protected. One possibility is to subclass Panel; another (which is what this program does) is to install an event handler for the panel's Resize event and invalidate the panel there.

Notice that the Font object created in the PanelOnPaint method uses a FontStyle argument stored as a field named fontstyle. This is the field that the buttons will alter.

As in the TextBoxWithToolBar program, the ToggleButtons program installs its ToolBarOnClick method as the event handler for the toolbar's ButtonClick event. The Style property for each button is set to ToolBarButtonStyle.ToggleButton in the constructor, and the ToolBarOnClick method assigns the Tag property for each button the FontStyle enumeration member associated with the button.
Once again, the use of an appropriate Tag property makes the ToolBarOnClick method relatively straightforward. After obtaining the ToolBarButton object from the event argument and checking that the Tag type is a FontStyle object, ToolBarOnClick sets or clears a bit in the fontStyle field.

```csharp
if (tbarbtn.Pushed)
    fontStyle ^= (FontStyle) tbarbtn.Tag;
else
    fontStyle &= ~(FontStyle) tbarbtn.Tag;
```

The method then concludes by invalidating the panel, which generates a call to PanelOnPaint.

Although I've demonstrated the PushButton and ToggleButton styles separately, keep in mind that any toolbar can contain a mix of buttons with different styles. The third style is ToolBarButtonStyle.DropDownButton, which invokes a menu indicated by the DropDownMenu property.

Although DropDownMenu is defined as an object of type Menu, it's really an object of type ContextMenu (which derives from Menu). You can define an appropriate object for the DropDownMenu property like so:

```csharp
ContextMenu menu = new ContextMenu();
menu.MenuItems.Add("First Item");
menu.MenuItems.Add("Second Item");
menu.MenuItems.Add("Third Item");
```

Of course, you'd also define event handlers for the items. Then assign these properties of the ToolBarButton object:

```csharp
tbarbtn.Style = ToolBarButtonStyle.DropDownMenu;
tbarbtn.DropDownMenu = menu;
```

A property of ToolBar indicates whether the drop-down buttons in the toolbar are displayed with little arrows that visually indicate that they invoke menus:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>DropDownArrows</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The default is true.

There's a catch to DropDownMenu buttons that I haven't mentioned yet. It's more of a convention than an actual requirement, but if you're familiar with applications that use such toolbar buttons, you know they don't invoke normal text-based menus. The menus instead usually contain little pictures. Thus, implementing a DropDownMenu button almost always involves an owner-draw menu.

Here's a program named DropDownMenuButton that derives from ToggleButtons and adds a fifth button to set the text color. The button invokes an owner-draw menu that displays 16 common colors and also re-creates the button image itself to indicate the selected color.
using System;
using System.Drawing;
using System.Windows.Forms;

class DropDownMenuButton: ToggleButtons
{
    public new static void Main()
    {
        Application.Run(new DropDownMenuButton());
    }
    public DropDownMenuButton()
    {
        Text = "Drop-Down Menu Button";
        strText = "Drop-Down";

        // Create bitmap for new button and add it to ImageList.
        tbar.ImageList.Images.Add(CreateBitmapButton(clrText));

        // Create the menu for the button.
        ContextMenu menu = new ContextMenu();

        EventHandler ehOnClick = new EventHandler(MenuColorOnClick);
        MeasureItemEventHandler ehOnMeasureItem =
            new MeasureItemEventHandler(MenuColorOnMeasureItem);
        DrawItemEventHandler ehOnDrawItem =
            new DrawItemEventHandler(MenuColorOnDrawItem);

        Color[] acolor =
            {
                Color.FromArgb(0x00, 0x00, 0x00), Color.FromArgb(0x00, 0x00, 0x80),
                Color.FromArgb(0x00, 0x80, 0x00), Color.FromArgb(0x00, 0x80, 0x80),
                Color.FromArgb(0x80, 0x00, 0x00), Color.FromArgb(0x80, 0x00, 0x80),
                Color.FromArgb(0x80, 0x80, 0x00), Color.FromArgb(0x80, 0x80, 0x80),
                Color.FromArgb(0xC0, 0xC0, 0xC0), Color.FromArgb(0x00, 0x00, 0xFF),
                Color.FromArgb(0x00, 0xFF, 0x00), Color.FromArgb(0x00, 0xFF, 0xFF),
            };
    }
for (int i = 0; i < acolor.Length; i++)
{
    MenuItemColor mic = new MenuItemColor();
    mic.OwnerDraw     = true;
    mic.Color         = acolor[i];
    mic.Click        += ehOnClick;
    mic.MeasureItem  += ehOnMeasureItem;
    mic.DrawItem     += ehOnDrawItem;
    mic.Break         = i % 4 == 0;

    menu.MenuItems.Add(mic);
}

// Finally, make the button itself.

ToolBarButton tbarbtn = new ToolBarButton();
tbarbtn.ImageIndex = 4;
tbarbtn.Style = ToolBarButtonStyle.DropDownButton;
tbarbtn.DropDownMenu = menu;
tbarbtn.ToolTipText = "Color";

tbar.Buttons.Add(tbarbtn);
}

void MenuColorOnClick(object obj, EventArgs ea)
{
    // Set the new text color.

    MenuItemColor mic = (MenuItemColor) obj;
    clrText = mic.Color;
    panel.Invalidate();

    // Make a new button bitmap.

    tbar.Invalidate();
}

void MenuColorOnMeasureItem(object obj, MeasureItemEventArgs miea)
{
    miea.ItemHeight = 18;
    miea.ItemWidth = 18;
}
void MenuColorOnDrawItem(object obj, DrawItemEventArgs diea) {
    MenuItemColor mic = (MenuItemColor) obj;
    Brush brush = new SolidBrush(mic.Color);

    Rectangle rect = diea.Bounds;

    rect.X += 1;
    rect.Y += 1;
    rect.Width -= 2;
    rect.Height -= 2;

    diea.Graphics.FillRectangle(brush, rect);
}

Bitmap CreateBitmapButton(Color clr)
{
    Bitmap bm = new Bitmap(16, 16);
    Graphics grfx = Graphics.FromImage(bm);
    Font font = new Font("Arial", 10, FontStyle.Bold);
    SizeF sizef = grfx.MeasureString("A", font);
    float fScale = Math.Min(bm.Width / sizef.Width,
                            bm.Height / sizef.Height);

    font = new Font(font.Name, fScale * font.SizeInPoints,
                    font.Style);
    StringFormat strfmt = new StringFormat();
    strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

    grfx.Clear(Color.White);
    grfx.DrawString("A", font, new SolidBrush(clr),
                    bm.Width / 2, bm.Height / 2, strfmt);
    grfx.Dispose();

    return bm;
}

class MenuItemColor: MenuItem
{
    Color clr;

    public Color Color
    {
        get { return clr; } 
        set { clr = value; } 
    }
}
This file also contains an override of the *MenuItem* class that stores a property named *Color*.

The *CreateBitmapButton* method toward the bottom of the program creates a 16-pixel-square bitmap containing a single “A” displayed with the Arial font and colored with the indicated argument to the method. The program's constructor begins by creating an initial bitmap and adding it to the *ImageList* object created by the ToggleButtons program.

The constructor then creates a context menu based on the *MenuItemColor* class defined at the bottom of the program. Event handlers for the *Click*, *MeasureItem*, and *DrawItem* events are installed. The button itself has a *Style* property of *ToolBarButtonStyle.DropDownButton*. The *DropDownMenu* property is set to the menu just created.

The *MeasureItem* event handler sets the size of the menu item to 18 pixels square, and the *DrawItem* handler draws a rectangle 1 pixel within the item bounds. Here's the resultant menu:

![Drop-Down Menu Button](image)

The images in the menu aren't square because the *DrawItem* event handler doesn't take account of the additional space added to the width for a possible check box.

The *MenuColorOnClick* event handler sets the *clrText* field that the *PanelOnPaint* method uses for coloring the text and then invalidates the *Panel*. It concludes by creating a new bitmap for the button.

One last warning: Because DropDownListButton inherits from ToggleButtons, and ToggleButtons loads a bitmap resource, you must change the resource namespace in DropDownListButton to "ToggleButtons" for the resource to be properly loaded. (I discussed this problem in Chapter 11.)

The standard toolbar allows push buttons, toggle buttons, and drop-down menus. If you'd like something more elaborate—a combo box, for example—you must take a slightly different approach. You'll want to create a *Panel* control that is a child of your *Form* and parent to one or more *ToolBar* controls and whatever *ComboBox* or other controls you need.
Chapter 21: Printing

Overview

Printing in a Windows Forms application is relatively painless, but the key word here is relatively. It seems easy only when you've had experience grappling with the Win32 printer API. Printing will never be quite as easy as displaying text and graphics on the screen, mostly because of the wide variety of printer types, their relatively slow speed, printer options (such as trays, bins, and paper sizes), and common problems such as paper jams.

Part of the difficulty in learning about printing in Windows Forms is the existence of several interlocking classes, all of which seem to refer to each other. For example, the PrinterSettings class has a property of type PageSettings, and the PageSettings class has a property of type PrinterSettings, and that's just the beginning. After awhile, the System.Drawing.Printing namespace starts to look like a hall of mirrors. Much of the process of learning about printing involves sorting out the various classes.

Although I'll be giving you enough information in the early pages of this chapter to handle printing entirely on your own, you'll probably want to take advantage of the common dialog box library (discussed toward the end of the chapter) to help out and make your application consistent with others. The System.Windows.Forms namespace contains classes to display standard print and page-setup dialog boxes, and a print preview window.

Let's begin this journey with the printers themselves.

Printers and Their Settings

Windows allows a user to install multiple printers. (More accurately, the user can install device drivers for multiple printers. The printers don't actually have to be attached to the machine.) The installed printers are listed in the Printers dialog box that you can invoke from the Settings item on the Start menu. At any time, only one of these printers is the default printer. You can change which printer is the default in this Printers dialog box.

From the perspective of a Windows Forms program, a particular printer is described by an object of type PrinterSettings, which like most of the classes I'll discuss in this chapter (except for the common dialog boxes) is defined in the System.Drawing.Printing namespace. The PrinterSettings class has only a default constructor, which creates an object for the default printer:

**PrinterSettings Constructor**

PrinterSettings ()

For example, the statement

PrinterSettings prnset = new PrinterSettings();

creates a new instance of PrinterSettings that refers to the default printer.

The following three properties indicate some basic information about the printer:

**PrinterSettings Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>PrinterName</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>IsValid</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IsDefaultPrinter</td>
<td>get</td>
</tr>
</tbody>
</table>

PrinterName is a string that usually indicates the make and model of the printer. It's the same string you'll see in the Printers dialog box. Here are some examples:
During the installation of a printer, the user can change the name that refers to the printer, so the
printer name you encounter in a PrinterSettings object might not be standard.

The IsValid and IsDefaultPrinter properties will usually be set to true when you create a new
PrinterSettings object. However, if no printers are installed, PrinterName returns the string "<no
default printer>" and IsValid equals false.

Notice that PrinterName is writable, which means that you can set it to a string that identifies another
installed printer. When you set the PrinterName property, all the other properties of PrinterSettings
also change to reflect that printer. Obviously, the string you set PrinterName to must match the name
of an installed printer. If the string doesn't match, no exception is thrown but the IsValid property will
be set to false.

To intelligently set the PrinterName property to the name of another installed printer, you can first
obtain a list of all installed printers by using the only static property of PrinterSettings:

<table>
<thead>
<tr>
<th>PrinterSettings Static Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>PrinterSettings.StringCollection</td>
</tr>
</tbody>
</table>

The StringCollection class is defined within PrinterSettings. It's really just an array of read-only
strings. Suppose you make use of the InstalledPrinters property like so:

PrinterSettings.StringCollection sc = PrinterSettings.InstalledPrinters();

You can then use the following two properties with the sc object:

<table>
<thead>
<tr>
<th>PrinterSettings.StringCollection Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>string</td>
</tr>
</tbody>
</table>

The quantity sc.Count is the number of installed printers (or 0 if no printers are installed), sc[0] is the
name of the first printer, sc[1] is the name of the second, and so forth.

You don't need to save the value of InstalledPrinters in a variable. You can access the property itself.
For example,

PrinterSettings.InstalledPrinters.Count

is the number of installed printers, and

PrinterSettings.InstalledPrinters[1]

is the name of the second printer. Here's some code that puts all the installed printers in a
ComboBox named combo:

foreach (string str in PrinterSettings.InstalledPrinters)
    combo.Items.Add(str);

You can change the printer that the PrinterSettings object refers to by setting the PrinterName
property to one of the strings in the collection. If you've defined the StringCollection variable sc, you
do it like this:

prnset.PrinterName = sc[2];

You can also assign the PrinterName property directly by indexing the InstalledPrinters property:
prnset.PrinterName = PrinterSettings.InstalledPrinters[2];

Unless something is seriously wrong, the IsValid property should then be true, and IsDefaultPrinter will be false, even if you set PrinterName to the name of the default printer.

Let me repeat: When you set the PrinterName property to the name of an installed printer, all the properties of the PrinterSettings object change to reflect that printer.

Here are a couple properties that indicate very basic capabilities of the printer:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>IsPlotter</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>SupportsColor</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>LandscapeAngle</td>
<td>get</td>
</tr>
</tbody>
</table>

If the IsPlotter property is true, you probably shouldn't rely on the printer to display bitmaps. If the printer doesn't support color, you may want to use alternatives to color in some graphics. (For example, if you're using color for bar graphs or maps, you may want to substitute hatch brushes when printing, as I discussed in Chapter 17.) The LandscapeAngle property usually indicates either 90 degrees or 270 degrees. However, if the printer isn't capable of landscape mode, the property will be equal to 0.

No further information is available from PrinterSettings about the technology of the printer (that is, whether it works by laser or ink jets or something else).

PrinterSettings also has three properties that return collections of items. These properties indicate the available paper sources on the printer (that is, bins and trays), the various paper sizes supported by the printer (including envelopes), and the display resolutions available on the printer:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrinterSettings.-</td>
<td>PaperSources</td>
<td>get</td>
<td>PaperSource</td>
</tr>
<tr>
<td>PaperSourceCollection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrinterSettings.-</td>
<td>PaperSizes</td>
<td>get</td>
<td>PaperSize</td>
</tr>
<tr>
<td>PaperSizeCollection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrinterSettings.PrinterResolutionCollection</td>
<td>PrinterResolutions</td>
<td>get</td>
<td>PrinterResolution</td>
</tr>
</tbody>
</table>

These properties are all quite similar in how they work. The three classes in the first column of the table (headed Type) are all defined within the PrinterSettings class, and each of them has just two, read-only properties: Count, which is the number of items in the collection, and an indexer, which returns an object of the type indicated in the last column of the table, labeled Items.

For example, the PaperSources property is essentially a collection of PaperSource objects. The quantity

prnset.PaperSources.Count

indicates the number of these PaperSource objects. You can reference each one by indexing the property, so that

prnset.PaperSources[2]

is an object of type PaperSource, the third in the collection. If the number of items in the collection is less than 3, then attempting to reference the third item in the array will cause an exception to be thrown.

Let's take a look at the PaperSource, PaperSize, and PrinterResolution classes. The PaperSource class has two, read-only properties:
### PaperSource Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>SourceName</td>
<td>get</td>
</tr>
<tr>
<td>PaperSourceKind</td>
<td>Kind</td>
<td>get</td>
</tr>
</tbody>
</table>

The `SourceName` property is a text description that should be meaningful to a user (such as "Manual Paper Feed"). `PaperSourceKind` is an enumeration:

### PaperSourceKind Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper</td>
<td>1</td>
<td>TractorFeed</td>
<td>8</td>
</tr>
<tr>
<td>Lower</td>
<td>2</td>
<td>SmallFormat</td>
<td>9</td>
</tr>
<tr>
<td>Middle</td>
<td>3</td>
<td>LargeFormat</td>
<td>10</td>
</tr>
<tr>
<td>Manual</td>
<td>4</td>
<td>LargeCapacity</td>
<td>11</td>
</tr>
<tr>
<td>Envelope</td>
<td>5</td>
<td>Cassette</td>
<td>14</td>
</tr>
<tr>
<td>ManualFeed</td>
<td>6</td>
<td>FormSource</td>
<td>15</td>
</tr>
<tr>
<td>AutomaticFeed</td>
<td>7</td>
<td>Custom</td>
<td>257</td>
</tr>
</tbody>
</table>

Keep in mind that the `PaperSources` property of `PrinterSettings` is a collection of all the possible paper sources on the printer. The property does not indicate the currently default paper source. (That's coming later.)

The `PaperSizes` property of `PrinterSettings` is a collection of all the paper sizes supported by the printer. Each item is an object of type `PaperSize`, which has the following four properties:

### PaperSize Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>PaperName</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Height</td>
<td>get/set</td>
</tr>
<tr>
<td>PaperKind</td>
<td>Kind</td>
<td>get</td>
</tr>
</tbody>
</table>

`PaperName` is a text string that should be meaningful to the user, such as "Envelope #10". The `Width` and `Height` properties indicate the size of the paper (or envelope) in hundredths of an inch. `PaperKind` is an enumeration that has more members (117 at last count) than is convenient to list here. Here are some sample values that might be encountered in the United States and Europe:

### PaperKind Enumeration (selection)

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter</td>
<td>1</td>
<td>8.5 in. by 11 in.</td>
</tr>
<tr>
<td>Legal</td>
<td>5</td>
<td>8.5 in. by 14 in.</td>
</tr>
<tr>
<td>Executive</td>
<td>7</td>
<td>7.25 in. by 10.5 in.</td>
</tr>
<tr>
<td>A4</td>
<td>9</td>
<td>210 mm by 297 mm</td>
</tr>
<tr>
<td>A5</td>
<td>11</td>
<td>148 mm by 210 mm</td>
</tr>
</tbody>
</table>

The `PrinterResolutions` property of `PrinterSettings` is a collection of `PrinterResolution` objects. The `PrinterResolution` class has three properties:

### PrinterResolution Properties
**PrinterResolutionKind** is another enumeration:

### PrinterResolutionKind Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>0</td>
</tr>
<tr>
<td>Draft</td>
<td>−1</td>
</tr>
<tr>
<td>Low</td>
<td>−2</td>
</tr>
<tr>
<td>Medium</td>
<td>−3</td>
</tr>
<tr>
<td>High</td>
<td>−4</td>
</tr>
</tbody>
</table>

Every printer has at least five items in the PrinterResolutions collection. Four of these items have **PrinterResolutionKind** values of Draft, Low, Medium, and High, with X and Y properties set equal to −1. These four enumeration values are not necessarily associated with unique printer resolutions. If the printer is capable of only one resolution, all these options result in the same resolution.

The remaining one or more items in the PrinterResolutions collection indicate the actual device resolutions available on the printer. These remaining items all have **PrinterResolutionKind** values of Custom. The X and Y properties indicate the actual resolution in dots per inch.

For example, a printer may be capable of two resolutions: 600 × 600 and 1200 × 1200. The PrinterResolutions collection will have six items. Two of the items will have **PrinterResolutionKind** values of Custom; one will have X and Y values of 600; the other will have X and Y values of 1200. The other four items are Draft, Low, Medium, and High with X and Y values of −1.

The following properties of PrinterSettings involve printing a multipage document:

### PrinterSettings Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>CanDuplex</td>
<td>get</td>
</tr>
<tr>
<td>Duplex</td>
<td>Duplex</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MaximumCopies</td>
<td>get</td>
</tr>
<tr>
<td>short</td>
<td>Copies</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>Collate</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The **CanDuplex** property is true if the printer is capable of printing on both sides of the page. If the property is true, you can set the **Duplex** property to one of the following values:

### Duplex Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplex</td>
<td>1</td>
</tr>
<tr>
<td>Vertical</td>
<td>2</td>
</tr>
<tr>
<td>Horizontal</td>
<td>3</td>
</tr>
<tr>
<td>Default</td>
<td>−1</td>
</tr>
</tbody>
</table>
The *Simplex* member indicates one-side printing. The *Vertical* and *Horizontal* options refer to the two different ways that double-sided pages can be printed. *Vertical* indicates that the pages are intended to be joined vertically, just like a regular book. The *Horizontal* option is for pages that are joined horizontally, usually at the top.

The *Copies* property is 1 by default. You can set it to any value up to *MaximumCopies* to force the printer driver to print multiple copies. *Collate* indicates the order of the copies. If you print two copies of three pages and *Collate* equals *false*, the pages will be printed in the order 1, 1, 2, 2, 3, 3. When *Collate* is *true*, the order is 1, 2, 3, 1, 2, 3. The default value of *Collate* depends on the printer.

If you programmatically set the following properties, nothing will happen. These properties are intended to be used in conjunction with the *PrintDialog* class that I'll discuss later in this chapter:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintRange</td>
<td>PrintRange</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MinimumPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MaximumPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>FromPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>ToPage</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>PrintToFile</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The final property of the *PrinterSettings* class is an object of type *PageSettings*, another important class in System.Drawing.Printing:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageSettings</td>
<td>DefaultPageSettings</td>
<td>get</td>
</tr>
</tbody>
</table>

I'll discuss the *PageSettings* class in detail shortly. The class describes the characteristics of a printed page. For example, *PrinterSettings* has a *PaperSources* property that is a collection of all the paper sources available on the printer. *PageSettings* has a *PaperSource* property that indicates a paper source for a particular page.

The *DefaultPageSettings* property in *PrinterSettings* indicates—as the name implies—the default page settings. As you'll see, you can change the page settings for an entire document or for each page as a document is being printed.

*PrinterSettings* has several methods that allow you to interface with Win32 code. In particular, you can copy the information from *PrinterSettings* into Win32 *DEVMODE* or *DEVNAMES* structures, or you can copy from a *DEVMODE* or *DEVNAMES* structure into *PrinterSettings*.

In addition, there is one method of *PrinterSettings* that might be of interest even for a Windows Forms program that isn't interfacing with Win32 code:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics</td>
<td>CreateMeasurementGraphics()</td>
</tr>
</tbody>
</table>

This method returns something that in Win32 is called an *information device context*. You can use the *Graphics* object from *CreateMeasurementGraphics* to obtain information about the printer, but not to draw on a printer page. This method allows you to obtain additional information about any installed printer at any time, for example, during a program's constructor. The ability to obtain such information is much less vital in Windows Forms, primarily because Windows Forms fonts are handled in a more device-independent manner than fonts in Win32 API programs.
Let's now move from `PrinterSettings` to `PageSettings`, and then we'll have conquered two of the basic classes of `System.Drawing.Printing`.

**Page Settings**

The `PageSettings` class describes those printer characteristics that can change with each page. It's tempting to consider a `PageSettings` object in a vacuum. However, a particular `PageSettings` object is always associated with a particular printer. A little thought will convince you why this is so: if the `PageSettings` object indicates that a page is to be printed on ledger paper (that's 17 inches by 11 inches), the printer better support that size.

Programs commonly get access to precreated `PageSettings` objects, such as the `DefaultPageSettings` property in `PrinterSettings`. But you can also create a `PageSettings` object using the class's constructor:

**PageSettings Constructors**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PageSettings()</code></td>
<td>Creates a <code>PageSettings</code> object for the default printer.</td>
</tr>
<tr>
<td><code>PageSettings(PrinterSettings prnset)</code></td>
<td>Creates a <code>PageSettings</code> object based on a particular installed printer indicated by the <code>PrinterSettings</code> argument.</td>
</tr>
</tbody>
</table>

In either case, the `PageSettings` object contains default page settings for the printer.

Default page settings for installed printers are defined by the user in Printing Preferences dialog boxes that the user invokes from the Printers dialog box. A Windows Forms program can change those default settings when printing a document, but any changes made by the Windows Forms program do not affect other applications. For example, if the user has selected landscape mode in the Printing Preferences dialog box, a Windows Forms program can print in portrait mode, but nothing the program does will change the landscape selection in the Printing Preferences dialog box.

The `PageSettings` class has eight properties, seven of which are writable as well as readable:

**PageSettings Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PrinterSettings</code></td>
<td><code>PrinterSettings</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>Landscape</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>Rectangle</code></td>
<td><code>Bounds</code></td>
<td><code>get</code></td>
</tr>
<tr>
<td><code>Margins</code></td>
<td><code>Margins</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>Color</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>PaperSource</code></td>
<td><code>PaperSource</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>PaperSize</code></td>
<td><code>PaperSize</code></td>
<td><code>get/set</code></td>
</tr>
<tr>
<td><code>PrinterResolution</code></td>
<td><code>PrinterResolution</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

Notice that the first property in this table is the `PrinterSettings` property that indicates the printer these page settings are associated with. When you obtain a `PageSettings` object from the `DefaultPageSettings` property of a `PrinterSettings` object, the `PrinterSettings` property of the `PageSettings` object is the same object as the original `PrinterSettings` object.

In other words, if you create a `PrinterSettings` object named `prnset`, then

```
(prnset == prnset.DefaultPageSettings.PrinterSettings)
```

returns `true`. Remember that objects are actually references, so any change you make to any property in `prnset` will be reflected in `prnset.DefaultPageSettings.PrinterSettings`.
However, if you create a `PageSettings` object named `pageset`, then

```csharp
(pageset == pageset.PrinterSettings.DefaultPageSettings)
```

returns `false`, even though all the corresponding properties of the two objects will initially be equal. An object of type `PageSettings` refers to the settings of a particular page. You may want to change the settings for a particular page without changing the default page settings for the document.

In most cases, you'll use the remaining properties in `PageSettings` just to obtain information. However, your program can also (within limits) set the properties to change the way in which a page is printed.

For example, the `Landscape` property is `false` to indicate portrait mode and `true` for landscape mode. That's informational. Your application can use that information to print somewhat differently depending on how the page is oriented. But your program can also change that property itself without any intervention by the user.

The read-only `Bounds` property is a `Rectangle` object that indicates the size of the page in units of hundredths of an inch, taking into account the paper size and the `Landscape` setting. For example, letter-size paper in portrait mode will have a `Bounds` property of (0, 0, 850, 1100). In landscape mode, the `Bounds` property is (0, 0, 1100, 850).

The `Margins` property indicates default margins for the page, which are initially set to 1 inch on all four sides. You can construct a new `Margins` object using the following constructors:

### Margins Constructors

```csharp
Margins()
Margins(int Left, int Right, int Top, int Bottom)
```

The class has four properties, which indicate the margins in hundredths of an inch:

### Margins Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Left</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Right</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Top</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Bottom</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Sometimes a user will specify that pages should not be printed in color even if the printer is capable of color. Perhaps the color ink-jet cartridge is empty. The `Color` property of the `PageSettings` object indicates whether the user wants color to be used on the page.

The next three properties in the table are `PaperSource`, `PaperSize`, and `PrinterResolution`. You'll recall that the `PrinterSettings` class has three properties named `PaperSources`, `PaperSizes`, and `PrinterResolutions` (all plurals) that correspond to these three properties of `PageSettings`. The `PaperSource` property in `PageSettings`, for example, is one of the items from the `PaperSources` collection in `PrinterSettings`.

If you want to change one of these three properties from your program, be sure to set the property from a member of the corresponding collection. For example, if you have an object of type `PageSettings` named `pageset` and you want to change the printer resolution to `Draft`, the code can look something like this:

```csharp
foreach (PrinterResolution prnres in
    pageset.PrinterSettings.PrinterResolutions)
{
    if (prnres.Kind == PrinterResolutionKind.Draft)
    {
```
The `foreach` statement loops through all the items of the `PrinterResolutions` collection in the `PrinterSettings` object associated with the `PageSettings` object. When there's a match, the code sets the `PrinterResolution` property. You need to set the `PrinterResolution` property of `PageSettings` from precreated `PrinterResolution` objects because the `PrinterResolution` class has no public constructor.

There aren't many occasions when your program will want to change the `PaperSource` or `PaperSize` property. However, suppose you implement a mail-merge facility and you want to alternately print letters and envelopes in one print job. You would need to change the `PaperSource` and `PaperSize` properties accordingly, based on specifications the user made in the application.

The `PaperSize` property is not affected by the `Landscape` property. If the `Landscape` property is `false`, the `Bounds` property `Width` and `Height` will equal the `Width` and `Height` properties of the `PaperSize` property. If `Landscape` is `true`, the `Bounds Width` and `Height` properties will be swapped. The `PaperSize` properties will not.

So far, we haven't gotten to the point where we can actually print something. That job requires defining an object of type `PrintDocument`.

### Defining a Document

A print job consists of one or more pages printed on a particular printer and is represented by the `PrintDocument` class. `PrintDocument` has only a default constructor:

```csharp
PrintDocument Constructor

PrintDocument()```

Generally, a program begins the process of printing by creating an object of type `PrintDocument`:

```csharp
PrintDocument prndoc = new PrintDocument();
```

You could create this object anew for each print job. However, if you're using the standard print dialog boxes—or some other means of allowing the user to select printers and printer options—you probably want to retain those settings in the `PrintDocument` object and use the same instance for the duration of the program. In that case, you'd define `prndoc` as a field and create it only once.

`PrintDocument` has only four properties, but two of them are objects of type `PrinterSettings` and `PageSettings`, so there's much more information packed into the `PrintDocument` object than you'd expect at first glance:

#### PrintDocument Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PrinterSettings</code></td>
<td><code>PrinterSettings</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>PageSettings</code></td>
<td><code>DefaultPageSettings</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>string</code></td>
<td><code>DocumentName</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>PrintController</code></td>
<td><code>PrintController</code></td>
<td>get/set</td>
</tr>
</tbody>
</table>

When you create a new `PrintDocument` object, the `PrinterSettings` property indicates the default printer. If you want, you can change the `PrinterSettings` property or individual properties of the `PrinterSettings` property. For example,

```csharp
prndoc.PrinterSettings.Copies = 2;
```

The `DefaultPageSettings` property is initially set from the `DefaultPageSettings` property of the `PrinterSettings` object. You can change that as well, or properties of that property, for example, as shown here:
prndoc.DefaultPageSettings.Landscape = true;

For a new PrintDocument object, the expression
prndoc.PrinterSettings == prndoc.DefaultPageSettings.PrinterSettings
returns true but
returns false. That's because you may want to change the DefaultPageSettings for the document without changing the default page settings for the printer.

The DocumentName property is initialized to the text string "document". You'll probably want to change that. The name shows up whenever the print job is identified, such as in the window that lists outstanding print jobs as they're being printed. I'll discuss the PrintController property shortly.

The PrintDocument class has four public events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginPrint</td>
<td>OnBeginPrint</td>
<td>PrintEventHandler</td>
<td>PrintEventArgs</td>
</tr>
<tr>
<td>QueryPageSettings</td>
<td>OnQueryPageSettings</td>
<td>QueryPageSettingsEventHandler</td>
<td>QueryPageSettingsEventArgs</td>
</tr>
<tr>
<td>PrintPage</td>
<td>OnPrintPage</td>
<td>PrintPageEventHandler</td>
<td>PrintPageEventArgs</td>
</tr>
<tr>
<td>EndPrint</td>
<td>OnEndPrint</td>
<td>PrintEventHandler</td>
<td>PrintEventArgs</td>
</tr>
</tbody>
</table>

The BeginPrint and EndPrint events are triggered once for every print job. The QueryPageSettings and PrintPage events are triggered for every page in the print job. The PrintPage event handler indicates whether there are more pages to be printed.

At the very least, you'll set a handler for the PrintPage event. If you want to use different page settings for each page (for example, to alternate the printing of letters and envelopes in a single print job), you'll also install a handler for the QueryPageSettings event. Install handlers for BeginPrint and EndPrint if you need to perform lengthy initialization or cleanup. I'll go over the arguments to these event handlers shortly.

Finally, you initiate printing by calling the following method, which is the only method in PrintDocument that isn't associated with an event:

PrintDocument Method

```csharp
void Print()
```

The Print method doesn't return until the program is finished printing the document. The application can't respond to any user input during this time. In the interim, the PrintDocument event handlers installed by the program will be called, beginning with the BeginPrint handler, then the QueryPageSettings and PrintPage handlers for each page, and finally the EndPrint handler.

Handling PrintDocument Events

The following class hierarchy shows the descendents of EventArgs that are involved with the PrintDocument event handlers:
CancelableEventArgs is defined in the System.ComponentModel namespace. The PrintEventArgs object associated with the BeginPrint and EndPrint events has a single property that it inherits from CancelEventArgs:

<table>
<thead>
<tr>
<th>PrintEventArgs Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

The BeginPrint event handler can set Cancel to true to abort the print job (for example, when the print job needs more memory than is available).

The QueryPageSettingsEventArgs class adds another property to Cancel:

<table>
<thead>
<tr>
<th>QueryPageSettingsEventArgs Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>PageSettings</td>
</tr>
</tbody>
</table>

The handler for the QueryPageSettings event can change PageSettings properties in preparation for the corresponding PrintPage event.

The PrintPageEventArgs class has four read-only properties and two read-write properties:

<table>
<thead>
<tr>
<th>PrintPageEventArgs Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Graphics</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>PageSettings</td>
</tr>
<tr>
<td>Rectangle</td>
</tr>
<tr>
<td>Rectangle</td>
</tr>
</tbody>
</table>

The Graphics object is created anew for each page. If you set properties of the Graphics object—such as PageUnit or PageScale—for one page, don’t expect the properties to be in effect for subsequent pages. The default PageUnit is GraphicsUnit.Display, which makes the printer seem like a 100-dpi device. The DpiX and DpiY properties of the Graphics object reflect the PrinterResolution property of PageSettings.

On entry to the print-page event handler, HasMorePages is always set to false. For printing multiple pages, you must set it to true on return from the event handler for the handler to be invoked again. On the last page, leave the property set to false.

The Cancel property is also usually set to false. Set it to true if your program needs to abort the print job. Setting Cancel to true is different from not setting HasMorePages to true in that the operating system will attempt to cease the printing of pages already in the queue.
The `PageSettings` property is for informational purposes while printing. The property will reflect any changes made in the `QueryPageSettings` event handler.

For your convenience, the `PrintPageEventArgs` object also includes a `PageBounds` rectangle, which is the same as the `Bounds` property of `PageSettings`, and a `MarginBounds` rectangle, which is the dimensions of the page less the margins indicated by the `Margins` property of `PageSettings`. I'll cover these properties in more detail shortly.

Let's take a look at some code. First, here's a simple dialog box that lets the user pick one of the installed printers from a combo box.

```csharp
class PrinterSelectionDialog : Form
{
    ComboBox combo;

    public PrinterSelectionDialog()
    {
        Text = "Select Printer";

        FormBorderStyle = FormBorderStyle.FixedDialog;
        ControlBox = false;
        MaximizeBox = false;
        MinimizeBox = false;
        ShowInTaskbar = false;
       StartPosition = FormStartPosition.Manual;
        Location = ActiveForm.Location +
            SystemInformation.CaptionButtonSize +
            SystemInformation.FrameBorderSize;

        Label label = new Label();
        label.Parent = this;
        label.Text = "Printer:";
        label.Location = new Point(8, 8);
        label.Size = new Size(40, 8);

        combo = new ComboBox();
        combo.Parent = this;
        combo.DropDownStyle = ComboBoxStyle.DropDownList;
        combo.Location = new Point(48, 8);
        combo.Size = new Size(144, 8);
    }
}
```
// Add the installed printers to the combo box.

foreach (string str in PrinterSettings.InstalledPrinters)
    combo.Items.Add(str);

Button btn = new Button();
btn.Parent = this;
btn.Text = "OK";
btn.Location = new Point(40, 32);
btn.Size = new Size(40, 16);
btn.DialogResult = DialogResult.OK;
AcceptButton = btn;

btn = new Button();
btn.Parent = this;
btn.Text = "Cancel";
btn.Location = new Point(120, 32);
btn.Size = new Size(40, 16);
btn.DialogResult = DialogResult.Cancel;
CancelButton = btn;

ClientSize = new Size(200, 56);
AutoScaleBaseSize = new Size(4, 8);
}

public string PrinterName
{
    set { combo.SelectedItem = value; }
    get { return (string) combo.SelectedItem; }
}
}

The combo box is of type *DropDownList*, so the user is prohibited from typing anything in the edit field. A read/write property named *PrinterName* allows a program to initialize the selected item in the combo box to the default printer and to obtain the item the user selects.

The PrintThreePages program makes use of this dialog box. To let the user initiate printing, the program creates a menu containing a File submenu with a Print item. The handler for the Print menu item displays the dialog box to let the user choose a printer. The program installs both *QueryPageSettings* and *PrintPage* event handlers to print three pages, each of which contains the page number in a big font centered on the page. Just for kicks, the program sets the resolution settings to "draft" mode for the entire document and alternates between portrait and landscape for each page.

*PrintThreePages.cs*

    //---------------------------------------------
    // PrintThreePages.cs © 2001 by Charles Petzold
    //---------------------------------------------
    using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintThreePages: Form {
    const int iNumberPages = 3;
    int iPageNumber;

    public static void Main()
    {
        Application.Run(new PrintThreePages());
    }

    public PrintThreePages()
    {
        Text = "Print Three Pages";
        Menu = new MainMenu();
        Menu.MenuItem.Add("&File");
        Menu.MenuItem[0].MenuItem.Add("&Print...",
            new EventHandler(MenuFilePrintOnClick));
    }

    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
        // Create PrintDocument.
        PrintDocument prndoc = new PrintDocument();

        // Create dialog box and set PrinterName property.
        PrinterSelectionDialog dlg = new PrinterSelectionDialog();
        dlg.PrinterName = prndoc.PrinterSettings.PrinterName;

        // Show dialog box and bail out if not OK.
        if (dlg.ShowDialog() != DialogResult.OK)
            return;

        // Set PrintDocument to selected printer.
        prndoc.PrinterSettings.PrinterName = dlg.PrinterName;

        // Set printer resolution to "draft".
        foreach (PrinterResolution prnres in
prndoc.PrinterSettings.PrinterResolutions)
{
    if (prnres.Kind == PrinterResolutionKind.Draft)
        prndoc.DefaultPageSSettings.PrinterResolution = prnres;
}

// Set remainder of PrintDocument properties.
prndoc.DocumentName = Text;
prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
prndoc.QueryPageSSettings += new QueryPageSSettingsEventHandler(OnQueryPageSSettings);

// Commence printing.
iPageNNumber = 1;
prndoc.Print();
}

void OnQueryPageSSettings(object obj, QueryPageSSettingsEventArgs qpsea)
{
    if (qpsea.PageSSettings.PrinterSettings.LandscapeAngle != 0)
        qpsea.PageSSettings.Landscape ^= true;
}

void OnPrintPage((object obj, PrintPageEventArgs ppea)
{
    Graphics grfx  = ppea.Graphics;
    Font     font  = new Font("Times New Roman", 360);
    string   str   = iPageNNumber.ToString();
    SizeF    sizef = grfx.MeasureString(str, font);
    grfx.DrawString(str, font, Brushes.Black,
        (grfx.VisibleClipBounds.Width - sizef.Width) / 2,
        (grfx.VisibleClipBounds.Height - sizef.Height) / 2);

    ppea.HasMorePages v= iPageNNumber < iNumberPages;
    iPageNNumber += 1;
}

Let's take a look at the MenuFilePrintOnClick method first. That's the method that's executed when the user selects Print from the File menu. It begins by creating a new PrintDocument object and a new PrinterSelectionDialog object. The constructor in PrinterSelectionDialog fills a combo box with installed printers. The method then sets the PrinterName property of the dialog box to the default printer:
dlg.PrinterName = prndoc.PrinterSettings.PrinterName;

That becomes the selected item in the combo box.
If the user returns from the dialog box by pressing OK, the PrinterName property of the PrinterSettings property of the PrintDocument object is set to the selected printer:

```csharp
prndoc.PrinterSettings.PrinterName = dlg.PrinterName;
```

The method then sets the PrinterResolution property of the DefaultPageSettings property of PrintDocument to draft mode using code similar to what I showed earlier for doing such jobs. All pages of the document will now be printed with the resolution associated with draft mode. (You can determine what that resolution is by examining the DpiX and DpiY properties of the Graphics object during the PrintPage method.)

The MenuFilePrintOnClick method concludes by setting the DocumentName property of the PrintDocument object, installing event handlers for the PrintPage and QueryPageSettings events, initializing the page number, and calling the Print method in PrintDocument to begin printing.

The next code executed in the program will be the OnQueryPageSettings event handler for the first page. If the printer supports landscape mode, the method toggles the Landscape property of the PageSettings object passed as a property of QueryPageSettingsEventArgs:

```csharp
qpsea.PageSettings.Landscape ^= true;
```

After the OnQueryPageSettings method returns, the OnPrintPage event handler is called for the first page. That handler displays a large number centered on the page.

If the printer is set up to print in portrait mode by default, the first page will be in landscape mode, the second in portrait, and the third in landscape. Notice that PrintPage doesn't have to do anything special except use the current VisibleClipBounds property of the Graphics object to center the text. VisibleClipBounds reflects the current orientation for the printer.

Whenever you print from PrintThreePages, the PrintDocument object is created anew. That means your default printer always shows up as the selected printer in the dialog box, even if you switched to another printer in a previous print job. You might want to consider storing the PrintDocument object as a field. Just move this entire statement outside the MenuFilePrintOnClick method:

```csharp
PrintDocument prndoc = new PrintDocument();
```

Now the object will retain any changes made while the program is running. As I mentioned earlier, no changes are made that affect any other application or the same application when run later.

**The Page Dimensions**

To intelligently draw text and graphics on a printer page, you need to know some details about the size of the area in which you can draw. From Chapter 5 until now, I've been assuming that you can draw anywhere on the printable area of the printer page. But you really should be drawing only within certain margins that the user has specified.

Unfortunately, taking into account the user's selection of margins is a problematic area of printing in a Windows Forms application. You may think you have all the information you need, but you really don't.

Let's take a look at what you do have. A PrintPage event handler is passed an object of type PrintPageEventArgs. One property of that class is a Rectangle object named PageBounds. PageBounds is equal to the Bounds property of the PageSettings class, and it indicates the dimension of the physical page, taking portrait or landscape orientation into account, in units of hundredths of an inch. For example, for 8.5-by-11-inch letter-size paper in portrait mode, PageBounds is equal to (0, 0, 850, 1100).

The PageSettings class also includes an object named Margins, which indicates the margins the user desires on all four sides of the page in units of hundredths of an inch. By default, all four margins are initially equal to 100.

The MarginBounds property of PrintPageEventArgs is a Rectangle object based on PageBounds but taking margins into account. For letter-size paper with default margins, MarginBounds is the rectangle (100, 100, 650, 900).

So far, so good. The problem, however, is that the Graphics object you obtain from PrintPageEventArgs is set up to print on the printable area of the page. Printers usually can't print to

```csharp
PageDimensions
```
the very edge of the paper because of the presence of rollers and paper guides and whatnot in the printer. The origin of this Graphics object—that is, the location where graphics appear when you specify the point \((0, 0)\) in drawing methods—is the upper left corner of the printable area of the page. The origin is consistent with the VisibleClipBounds property of the Graphics object.

When my printer is loaded with standard 8.5-by-11-inch paper and printing is set for portrait mode, VisibleClipBounds reports a rectangle of \((0, 0, 800, 1060)\). By default, these units are 1/100 inch, so the printable area of the page is 8 inches wide and 10.6 inches high. The unprintable area is 0.5 inch total on the left and right, and 0.4 inch total on the top and bottom.

However, you can't assume that the unprintable area on my printer is 0.25 inch on the left and right, and 0.20 inch on the top and bottom. Depending on the printer, the unprintable area might be unequally distributed between the left and right, and the top and bottom of the page.

What we need is a rectangle describing the printable area of the page relative to the total page. Unfortunately, we've now run out of information. No more information exists in PrinterSettings, PageSettings, PrintDocument, or PrintPageEventArgs that reveals how the unprintable area is distributed along the edges of the page.

If you're content to live with approximations, you can calculate a rectangle relative to VisibleClipBounds (and hence usable with Graphics drawing methods) that describes the area of the page within the user-selected margins. If the PrintPageEventArgs object is named \texttt{ppea} and the Graphics object is named \texttt{grfx}, the expression

\[
\frac{\text{ppea.PageBounds.Width} - \text{grfx.VisibleClipBounds.Width}}{2}
\]

is the approximate unprintable margin on the left side of the page, and

\[
\frac{\text{ppea.PageBounds.Height} - \text{grfx.VisibleClipBounds.Height}}{2}
\]

is the approximate unprintable margin on the top side of the page. Subtract these two values from \texttt{ppea.MarginBounds.Left} and \texttt{ppea.MarginBounds.Top}, respectively, and you get the point in drawing coordinates that is approximately the upper left corner of the area of the page in which you should print to respect the user's margins.

Here's a calculation of a display rectangle that takes the user's margins into account:

```csharp
RectangleF rectf = new RectangleF(
    ppea.MarginBounds.Left -
    (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
    ppea.MarginBounds.Top -
    (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2,
    ppea.MarginBounds.Width, ppea.MarginBounds.Height);
```

Again, let me emphasize that this is an approximate calculation because it assumes the unprintable margins are distributed equally between the left and right sides, and the top and bottom. But it's the best you can do within the Windows Forms interface.

I use this rectangle calculation in the following program, which simply draws the rectangle and two lines connecting its corners.

```
PrintWithMargins.cs

 преимущество программы, которая просто рисует прямоугольник и две прямые линии, соединяющие его углы.

PrintWithMargins.cs

//-----------------------------------------------
// PrintWithMargins.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintWithMargins: Form
```
public static void Main()
{
    Application.Run(new PrintWithMargins());
}

public PrintWithMargins()
{
    Text = "Print with Margins";
    Menu = new MainMenu();
    Menu.MenuItems.Add("&File");
    Menu.MenuItems[0].MenuItems.Add("&Print...",
        new
        EventHandler(MenuFilePrintOnClick));
}

void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    // Create PrintDocument.
    PrintDocument prndoc = new PrintDocument();

    // Create dialog box and set PrinterName property.
    PrinterSelectionDialog dlg = new PrinterSelectionDialog();
    dlg.PrinterName = prndoc.PrinterSettings.PrinterName;

    // Show dialog box and bail out if not OK.
    if (dlg.ShowDialog() != DialogResult.OK)
        return;

    // Set PrintDocument to selected printer.
    prndoc.PrinterSettings.PrinterName = dlg.PrinterName;

    // Set remainder of PrintDocument properties and commence.
    prndoc.DocumentName = Text;
    prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
    prndoc.Print();
}

void OnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;
    RectangleF rectf = new RectangleF(
        ppea.MarginBounds.Left -
        (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
If you're not satisfied with this approximation, you'll have to access the Win32 `GetDeviceCaps` function with the arguments `PHYSICALOFFSETX` and `PHYSICALOFFSETY`.

If you're more comfortable with units other than 1/100 inch, you can convert the `PageBounds` and `MarginBounds` values to something else using the `PrinterUnitConvert` class. This class has one static method, named `Convert`, that's defined in six versions:

**PrinterUnitConvert Static Convert Method**

```csharp
int Convert(int iValue, PrinterUnit puFrom, PrinterUnit puTo)
double Convert(double dValue, PrinterUnit puFrom, PrinterUnit puTo)
Point Convert(Point pt, PrinterUnit puFrom, PrinterUnit puTo)
Size Convert(Size size, PrinterUnit puFrom, PrinterUnit puTo)
Rectangle Convert(Rectangle rect, PrinterUnit puFrom, PrinterUnit puTo)
Margins Convert(Margins margins, PrinterUnit puFrom, PrinterUnit puTo)
```

`PrinterUnit` is an enumeration:

**PrinterUnit Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>0</td>
</tr>
<tr>
<td>ThousandthsOfAnInch</td>
<td>1</td>
</tr>
<tr>
<td>HundredthsOfAMillimeter</td>
<td>2</td>
</tr>
<tr>
<td>TenthsOfAMillimeter</td>
<td>3</td>
</tr>
</tbody>
</table>

The member `Display` indicates hundredths of an inch.

**The Print Controller**

Earlier, when discussing the `PrintDocument` class, I skipped over the `PrintController` property. By default, you can set that property to an instance of a class descended from the abstract `PrintController` class. Here's the class hierarchy:
The *PrintController* class defines four methods:

**PrintController Methods**

```csharp
void OnStartPrint(PrintDocument prndoc, PrintEventArgs pea)
Graphics OnStartPage(PrintDocument prndoc, PrintPageEventArgs ppea)
void OnEndPage(PrintDocument prndoc, PrintPageEventArgs ppea)
void OnEndPrint(PrintDocument prndoc, PrintEventArgs pea)
```

As you've seen, when a program initiates printing by calling the *Print* method of the *PrintDocument* class, the *PrintDocument* object responds by triggering the four events defined by the class. These events are *BeginPrint*, *QueryPageSettings*, *PrintPage*, and *EndPrint*.

But *PrintDocument* also makes calls to the four methods of the particular *PrintController* object that is indicated by its *PrintController* property. *PrintDocument* calls the *OnStartPrint* method of *PrintController* after triggering its own *BeginPrint* event. *PrintDocument* calls *OnStartPage* and *OnEndPage* before and after triggering each *PrintPage* event. And finally, *PrintDocument* calls *OnEndPrint* in the *PrintController* after triggering its own *EndPrint* event.

In particular, the *OnStartPage* method in the *PrintController* object is responsible for obtaining the *Graphics* object that is eventually passed to the *PrintPage* event handler. (Notice the return value from the *OnStartPage* method.) This *Graphics* object essentially determines where the graphics output in *PrintPage* goes.

Normally, the graphics output goes to the printer, of course, and that's the responsibility of the *PrintController* object. However, the *PreviewPrintController* object has something else in mind. This particular controller creates a *Graphics* object based on a bitmap that represents the printer page. And that (as we shall see toward the end of this chapter) is how print preview is implemented in Windows Forms.

The default *PrintController* property of *PrintDocument* is an object of type *PrintControllerWithStatusDialog*, and that very name discloses another responsibility of the print controller: it displays the dialog box that shows the name of the print document and the page currently printing.

If you don't want that dialog box to be displayed, set the *PrintController* property of *PrintDocument* to an object of type *StandardPrintController*. *StandardPrintController* does everything *PrintControllerWithStatusDialog* does but without the dialog box.

If you'd prefer to display printing progress with something other than the dialog box, you can derive a class from *StandardPrintController*. Here, for example, is a print controller that displays the printing status in a panel of a status bar.

**StatusBarPrintController.cs**

```
//-------------------------------------------------------
// StatusBarPrintController.cs © 2001 by Charles Petzold
//-------------------------------------------------------

using System;

```
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class StatusBarPrintController : StandardPrintController
{
    StatusBarPanel statpanel;
    int iPageNumber;
    string strSaveText;

    public StatusBarPrintController(StatusBarPanel sbp): base()
    {
        statpanel = sbp;
    }
    public override void OnStartPrint(PrintDocument prndoc,
                                       PrintEventArgs pea)
    {
        strSaveText = statpanel.Text;     // Probably "Ready" or
                                        // similar
                                        statpanel.Text = "Starting printing";
        iPageNumber = 1;

        base.OnStartPrint(prndoc, pea);
    }
    public override Graphics OnStartPage(PrintDocument prndoc,
                                          PrintPageEventArgs ppea)
    {
        statpanel.Text = "Printing page " + iPageNumber++;

        return base.OnStartPage(prndoc, ppea);
    }
    public override void OnEndPage(PrintDocument prndoc,
                                    PrintPageEventArgs ppea)
    {
        base.OnEndPage(prndoc, ppea);
    }
    public override void OnEndPrint(PrintDocument prndoc,
                                     PrintEventArgs pea)
    {
        statpanel.Text = strSaveText;

        base.OnEndPrint(prndoc, pea);
    }
}
Notice first that the class overrides all four methods of StandardPrintController but also makes sure to call the corresponding methods in the base class. Doing so assures that all the normal activity of the print controller still takes place. The only enhancement this version adds is to keep a status bar panel updated. The panel object is required in the class's constructor.

Here's a version of PrintThreePages that forgoes the dialog box to select a printer but instead creates a status bar with one panel.

**PrintWithStatusBar.cs**

```
//-------------------------------------------------
// PrintWithStatusBar.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintWithStatusBar: Form
{
    StatusBar     sbar;
    StatusBarPanel sbarpanel;

    const int iNumberOfPages = 3;
    int     iPageNumber;

    public static void Main()
    {
        Application.Run(new PrintWithStatusBar());
    }

    public PrintWithStatusBar()
    {
        Text = "Print with Status Bar";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Print",
               new EventHandler(MenuFilePrintOnClick));

        sbar = new StatusBar();
        sbar.Parent = this;
        sbar.ShowPanels = true;

        sbarpanel = new StatusBarPanel();
        sbarpanel.Text = "Ready";
        sbarpanel.Width = Width / 2;
        sbar.Panels.Add(sbarpanel);
    }

    void MenuFilePrintOnClick(object obj, EventArgs ea)
```
When setting up the `PrintDocument` in response to the menu click, this version also sets the `PrintController` property:

```csharp
prndoc.PrintController = new StatusBarPrintController(statpanel);
```

Because the three pages went by a little too fast for me to confirm that the program was working right, I inserted a call to the `Sleep` method of the `Thread` class in the `OnPrintPage` method.

The only problem with this approach to displaying printer status is that it's deceptive. The absence of the modal dialog box indicates to the user that the application is ready to continue responding to user input. It's not. The application can't respond to user input until the `Print` method of `PrintDocument` returns. When a program reports printing status in a status bar, it should also implement background printing, which requires a second thread of execution.

**Using the Standard Print Dialog Box**

Part of the collection of common dialog boxes in `System.Windows.Forms` is `PrintDialog`, a dialog box that lets users select a printer and change the settings for that printer. `PrintDialog` also includes a facility for users to specify whether to print an entire document, a range of pages, or a selection (that is, the part of the document that has been highlighted).

You create a new `PrintDialog` object with the default constructor:
You must initialize one (but not both) of the following properties:

**PrintDialog Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintDocument</td>
<td>Document</td>
<td>get/set</td>
</tr>
<tr>
<td>PrinterSettings</td>
<td>PrinterSettings</td>
<td>get/set</td>
</tr>
</tbody>
</table>

Setting the Document property is preferred; the PrintDialog object then uses the PrinterSettings property from that PrintDocument object to set its own PrinterSettings property.

The bulk of the additional options available with PrintDialog involve letting the user print an entire document, a range of pages, or the current selection. The PrintDialog dialog box displays these three options (labeled All, Pages, and Selection) as radio buttons. By default, only the All option is enabled and, of course, it’s checked.

You can optionally enable the Pages and Selection buttons as well. You do so (and select a few other options on the dialog box) using the following properties:

**PrintDialog Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>AllowSelection</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>AllowSomePages</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>AllowPrintToFile</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>PrintToFile</td>
<td>get/set</td>
<td>false</td>
</tr>
<tr>
<td>bool</td>
<td>ShowNetwork</td>
<td>get/set</td>
<td>true</td>
</tr>
<tr>
<td>bool</td>
<td>ShowHelp</td>
<td>get/set</td>
<td>false</td>
</tr>
</tbody>
</table>

If you set ShowHelp to true, you must install a handler for the HelpRequest event (inherited from CommonDialog). The AllowPrintToFile property enables the check box for printing to a file. The PrintToFile property indicates whether or not the check box is checked.

When you enable the Pages radio button, the user has the option to type in a From page and a To page. You can specify initial values and minimum and maximum values for these two fields, but not as properties in PrintDialog. The properties are defined instead in PrinterSettings. After you set the Document property of PrintDialog, you can use the PrinterSettings property of PrintDialog to reference these properties:

**PrinterSettings Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintRange</td>
<td>PrintRange</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MinimumPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MaximumPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>FromPage</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>ToPage</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>PrintToFile</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The PrintRange property is an enumeration of type PrintRange, which has the following values:

**PrintRange Enumeration**
You’re probably tempted to set *MinimumPage* to 1 and *MaximumPage* to the total number of pages in the document. You can also initialize *FromPage* and *ToPage* to those same values. That’s actually not such a hot idea for some applications (for example, a Notepad clone). When the *PrintDialog* dialog box is displayed, the user has the option to change the printer, orientation, page size, and so forth, and any of these items could affect the total number of pages in the printed document.

As with any common dialog box, after initializing the *PrintDialog* object, you call its *ShowDialog* method. *ShowDialog* returns a *DialogResult* enumeration value. On return from *PrintDialog*, the *PrintRange* property indicates which option the user has selected. For a range of pages, *FromPage* and *ToPage* indicate the page range.

Let’s start off simple. The following program is another version of PrintThreePages that uses a *PrintDialog* object.

*SimplePrintDialog.cs*

```csharp
//------------------------------------------------
// SimplePrintDialog.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class PrintDialogHelloWorld: Form
{
    const int iNumberPages = 3;
    int iPagesToPrint, iPageNNumber;

    public static void Main()
    {
        Application.Run(new PrintDialogHelloWorld());
    }
    public PrintDialogHelloWorld()
    {
        Text = "Simple PrintDialog";

        Menu = new MainMenu();
        Menu.MenuItems.Add("&File");
        Menu.MenuItems[0].MenuItems.Add("&Print...",
            new EventHandler(MenuFilePrintOnClick));
    }
    void MenuFilePrintOnClick(object obj, EventArgs ea)
    {
```
// Create the PrintDocument and PrintDialog.

PrintDocument prndoc = new PrintDocument();
PrintDialog prndlg = new PrintDialog();
prndlg.Document = prndoc;

// Allow a page range.

prndlg.AllowSomePages = true;
prndlg.PrinterSettings.MinimumPage = 1;
prndlg.PrinterSettings.MaximumPage = iNumberPages;
prndlg.PrinterSettings.FromPage = 1;
prndlg.PrinterSettings.ToPage = iNumberPages;

// If the dialog box returns OK, print.

if (prndlg.ShowDialog() == DialogResult.OK)
{
    prndoc.DocumentName = Text;
    prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);

    // Determine which pages to print.

    switch (prndlg.PrinterSettings.PrintRange)
    {
        case PrintRange.AllPages:
            iPagesToPrint = iNumberPages;
            iPageNNumber = 1;
            break;

        case PrintRange.SomePages:
            iPagesToPrint = 1 + prndlg.PrinterSettings.ToPage - prndlg.PrinterSettings.FromPage;
            iPageNNumber = prndlg.PrinterSettings.FromPage;
            break;
    }
    prndoc.Print();
}

void OnPrintPage((object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;
    Font font = new Font("Times New Roman", 360);
    string str = iPageNNumber.ToString();
    SizeF sizef = grfx.MeasureString(str, font);
The hard part of this program is allowing the user to select a range of pages to print. The
\texttt{iNumberPages} field is hard-coded as 3. But along with the \texttt{iPageNumber} field, I also include a new
field, named \texttt{iPagesToPrint}, which can take on a value from 1 to \texttt{iNumberPages}.

Before invoking \texttt{PrintDialog}, the program sets all the properties of \texttt{PrintDialog} (including the
properties in \texttt{PrinterSettings} rather than \texttt{PrintDialog} itself) that involve the page range. On returning
from \texttt{PrintDialog}, the program checks the value of \texttt{PrintRange} and initializes \texttt{iPagesToPrint} and
\texttt{iPageNumber} (the first page to print) accordingly.

Some features of \texttt{PrintDialog} are handled for you automatically. If you select Print To File, a dialog
box appears asking you for a filename, and the graphics output for the printer is saved in that file.

As you can see by experimentation, every time \texttt{PrintDialog} is invoked, all the settings revert to their
default values. As you probably know by experience, some Windows applications also work like that,
and they can be quite annoying. If you have multiple printers, for example, you may want to print on
a nondefault printer from a particular application. If you select that nondefault printer once in
\texttt{PrintDialog}, the dialog box should probably display that printer on subsequent evocations. The same
goes for other settings you specify in the dialog box.

To make settings persistent, store both the \texttt{PrintDialog} and \texttt{PrintDocument} objects as fields. You can
simply move both these statements outside the \texttt{MenuFilePrintOnClick} method:

\begin{verbatim}
PrintDocument prndoc = new PrintDocument();
PrintDialog   prndlg = new PrintDialog();
\end{verbatim}

Then move this statement to the constructor:

\begin{verbatim}
prndlg.Document = prndoc;
\end{verbatim}

I'll be using this approach in the remaining programs in this chapter.

\section*{Setting Up the Page}

The second common dialog box connected with printing is \texttt{PageSetupDialog}. This dialog box usually
lets the user specify margins, page orientation, page sources, and paper sizes. But the dialog box
can also be used to select a default printer and printer options. \texttt{PageSetupDialog} has a single
constructor:

\begin{verbatim}
PageSetupDialog Constructor

PageSetupDialog()
\end{verbatim}

You then set one (and only one) of the following properties:

\begin{verbatim}
PageSetupDialog Properties (selection)
\end{verbatim}

\begin{tabular}{|l|l|l|}
\hline
Type & Property & Accessibility \\
\hline
PrintDocument & Document & get/set \\
\hline
\end{tabular}
Setting the `Document` property is recommended. `PageSetupDialog` then sets the `PrinterSettings` and `PageSettings` properties from that `PrintDocument` object. To make everything work right, you must use the same `PrintDocument` object with both `PageSetupDialog` and `PrintDialog`.

Here are the remaining `PageSetupDialog` properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>AllowMargins</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>AllowOrientation</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>AllowPaper</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>AllowPrinter</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>ShowNetwork</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>ShowHelp</td>
<td>get/set</td>
</tr>
<tr>
<td>Margins</td>
<td>MinMargins</td>
<td>get/set</td>
</tr>
</tbody>
</table>

All the `bool` properties are `true` by default except `ShowHelp`. Setting them to `false` disables certain aspects of the dialog box. The Network button is on the additional dialog box invoked when you press the Printer button. The `MinMargins` property is set to all zeros by default.

Changes the user makes in the `PageSetupDialog` dialog box are reflected in the `PageSettings` object that the dialog box obtains from the `PrintDocument` object.

The `ImagePrint` program derives from `ImageIO` in Chapter 16 (which itself derived from `ImageOpen`) to add Page Setup and Print options to the File menu. You can now use the program to load, save, and print bitmaps.

`ImagePrint.cs`

```csharp
//-----------------------------------------
// ImagePrint.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class ImagePrint: ImageIO
{
    PrintDocument prndoc = new PrintDocument();
    PageSetupDialog setdlg = new PageSetupDialog();
    PrintDialog prndlg = new PrintDialog();

    MenuItem miFileSet, miFilePrint, miFileProps;
```
public new static void Main()
{
    Application.Run(new ImagePrint());
}
public ImagePrint()
{
    Text = strProgName = "Image Print";

    // Initialize PrintDocument and common dialog boxes.
    prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
    setdlg.Document = prndoc;
    prndlg.Document = prndoc;

    // Add menu items.
    Menu.MenuItems[0].Popup += new EventHandler(MenuFileOnPopup);
    Menu.MenuItems[0].MenuItems.Add("-");
    // File Page Setup item
    miFileSet = new MenuItem("Page Setup...");
    miFileSet.Click += new EventHandler(MenuFileSetupOnClick);
    Menu.MenuItems[0].MenuItems.Add(miFileSet);

    // File Print item
    miFilePrint = new MenuItem("Print...");
    miFilePrint.Click += new EventHandler(MenuFilePrintOnClick);
    miFilePrint.Shortcut = Shortcut.CtrlP;
    Menu.MenuItems[0].MenuItems.Add(miFilePrint);
    Menu.MenuItems[0].MenuItems.Add("-");

    // File Properties item
    miFileProps = new MenuItem("Properties...");
    miFileProps.Click += new EventHandler(MenuFilePropsOnClick);
    Menu.MenuItems[0].MenuItems.Add(miFileProps);
}
protected override void OnPaint(PaintEventArgs pea)
{
    if (image != null)
        ScaleImageIsotropically(pea.Graphics, image, ClientRectangle);
}
void MenuFileOnPopup(object obj, EventArgs ea)
miFileSet.Enabled =
miFilePrint.Enabled =
miFileProps.Enabled = (image != null);
}
void MenuFileSetupOnClick(object obj, EventArgs ea)
{
    setdlg.ShowDialog();
}
void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    if (prndlg.ShowDialog() == DialogResult.OK)
    {
        prndoc.DocumentName = Text;
        prndoc.Print();
    }
}
void MenuFilePropsOnClick(object obj, EventArgs ea)
{
    string str =
        "Size = " + image.Size +
        "\nHorizontal Resolution = " + image.HorizontalResolution +
        "\nVertical Resolution = " + image.VerticalResolution +
        "\nPhysical Dimension = " + image.PhysicalDimension +
        "\nPixel Format = " + image.PixelFormat;
    MessageBox.Show(str, "Image Properties");
}
void OnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;
    RectangleF rectf = new RectangleF(
        ppea.MarginBounds.Left -
        (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width) / 2,
        ppea.MarginBounds.Top -
        (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height) / 2,
        ppea.MarginBounds.Width,
        ppea.MarginBounds.Height);
    ScaleImageIsotropically(grfx, image, rectf);
}
void ScaleImageIsotropically(Graphics grfx, Image image,
    RectangleF rectf)
{
    SizeF sizef = new SizeF(image.Width /
        image.HorizontalResolution,
float fScale = Math.Min(rectf.Width / sizef.Width, 
rectf.Height / sizef.Height);

sizef.Width *= fScale;
sizef.Height *= fScale;

grfx.DrawImage(image, rectf.X + (rectf.Width - sizef.Width) / 
2, 
rectf.Y + (rectf.Height - sizef.Height) / 
2, 
sizef.Width, sizef.Height);

PrintDocument and both dialog boxes (PageSetupDialog and PrintDialog) are defined as fields and initialized in the program's constructor. The constructor also adds Page Setup and Print items to the menu as well as a Properties item that displays information about the image.

Processing the Page Setup menu item is a snap. The MenuFileSetupOnClick method simply calls the ShowDialog method of PageSetupDialog. It doesn't even have to check the return value. The Print menu item is also fairly simple: The MenuFilePrintOnClick method begins by invoking PrintDialog. If PrintDialog returns DialogResult.OK, the method sets the DocumentName property of the PrintDocument object and calls Print. (The handler for the PrintPage event is set during the constructor.)

The OnPrintPage method first calculates a display rectangle with margins using the formulas I showed in the PrintWithMargins program. The method then displays the bitmap as large as possible within these margins while maintaining the correct aspect ratio. It makes use of a slightly modified version of the ScaleImageIsotropically method in the ImageScaleIsotropic program from Chapter 11. (I changed the Rectangle argument in the original version to a RectangleF to be consistent with the calculated display rectangle.) The program also overrides the previous OnPaint method to use the same display logic.

Print Preview

Once your application supports printing, it's fairly easy to implement a print preview feature. Basically, your normal PrintPage event handler is used to display printer output on the surfaces of bitmaps rather than to the printer. These bitmaps are then presented to the user. But before I show you how easy it is, let's examine what goes on behind the scenes. You may want to know these details if you prefer to take a different approach to handling the print preview bitmaps.

The key to print preview is the PrintController property of PrintDocument. By default, PrintController is set to PrintControllerWithStatusDialog, but I've already demonstrated how you can change this property to something else to allow you to display an alternative to the status dialog box.

For a more extreme effect, you can set the PrintController property of PrintDocument to an object of type PreviewPrintController:

PreviewPrintController ppc = new PreviewPrintController();
prndoc.PrintController = ppc;

The PreviewPrintController class has a single property:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>UseAntiAlias</td>
<td>get/set</td>
</tr>
</tbody>
</table>
Set the other properties of the `PrintDocument` object as usual, just as if you were going to print. Then initiate printing normally by calling the `Print` method of `PrintDocument`:

```csharp
prndoc.Print();
```

Recall that the print controller is responsible for obtaining the `Graphics` object that the `PrintDocument` passes to the `PrintPage` event handler. `PreviewPrintController` doesn't obtain a `Graphics` object for the printer. Instead, it creates a bitmap for each page and obtains a `Graphics` object for drawing on that bitmap. That's actually the `Graphics` object passed to the `PrintPage` event handler.

When the `Print` method returns, you can get access to these bitmaps. The only noninherited method of `PreviewPrintController` returns an array of `PreviewPageInfo` classes:

```csharp
PreviewPrintController.Method

PreviewPageInfo[] GetPreviewPageInfo()
```

The `PreviewPageInfo` class has two properties:

```
PreviewPageInfo Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Image</td>
<td>get</td>
</tr>
<tr>
<td>Size</td>
<td>PhysicalSize</td>
<td>get</td>
</tr>
</tbody>
</table>
```

The pixel size of the `Image` property is the pixel size of your printer page. The `PhysicalSize` property indicates the dimensions in hundredths of an inch. Now you have a collection of bitmaps, each of which corresponds to a page of the printed document. You can display these bitmaps however you want.

Is there an even easier approach? Yes, there is. You begin by creating an object of type `PrintPreviewDialog`:

```csharp
PrintPreviewDialog predlg = new PrintPreviewDialog();
```

`PrintPreviewDialog` is descended from `Form`, so it has lots of properties, methods, and events. But you really don't have to bother with many of them. Here are a few that `PrintPreviewDialog` implements itself:

```
PrintPreviewDialog Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintDocument</td>
<td>Document</td>
<td>get/set</td>
</tr>
<tr>
<td>PrintPreviewControl</td>
<td>PrintPreviewControl</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>UseAntiAlias</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>HelpButton</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

The essential property you must set is `Document`, and you set it to the same `PrintDocument` object you use for printing and page setup. `PrintPreviewControl` is another class defined in `System.Windows.Forms` and represents the controls that ultimately appear on the form that displays the bitmaps with your page images.

Here's the usual code to initialize and initiate print preview:

```csharp
predlg.Document = prndoc;
predlg.ShowDialog();
```

The `ShowDialog` method does all the work. It takes the `PrintDocument` object stored as its `Document` property, sets the `PrintController` to `PreviewPrintController`, calls the `Print` method of `PrintDocument`, and then shows a form displaying those bitmaps with a series of controls. You can also print from
print preview, in which case the print preview dialog box just uses the same PrintDocument object with the same event handlers. For that eventuality, you should set the DocumentName property of PrintDocument before calling ShowDialog.

Let's implement printing with page setup and print preview in our Notepad clone. The NotepadCloneWithPrinting program inherits from the NotepadCloneWithFormat program in Chapter 18, "Edit, List, and Spin."

**NotepadCloneWithPrinting.cs**

```csharp
//-------------------------------------------------------
// NotepadCloneWithPrinting.cs © 2001 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.IO;
using System.Windows.Forms;

class NotepadCloneWithPrinting: NotepadCloneWithFormat
{
    PrintDocument     prndoc = new PrintDocument();
    PageSSetupDialog   setdlg = new PageSSetupDialog();
    PrintPreviewDialog predlg = new PrintPreviewDialog();
    PrintDialog        prndlg = new PrintDialog();
    string             strPrintText;
    int                iStartPage, iNumPagesi, iPageNNumber;

    public new static void Main()
    {
        System.Threading.ApartmentState.STA;

        Application.Run(new NotepadCloneWithPrinting());
    }
    public NotepadCloneWithPrinting()
    {
        strProgName = "Notepad Clone with Printing";
        MakeCaption();

        prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);v;
        setdlg.Document = prndoc;
        predlg.Document = prndoc;
        prndlg.Document = prndoc;

        prndlg.AllowSomePages = true;
        prndlg.PrinterSettings.FromPage = 1;
        prndlg.PrinterSettings.ToPage =
```
protected override void MenuFileSetupOnClick(object obj, EventArgs ea)
{
    setdlg.ShowDialog();
}

protected override void MenuFilePreviewOnClick(object obj, EventArgs ea)
{
    prndoc.DocumentName = Text; // Just in case it's printed

    strPrintText = textbox.Text;
    iStartPage   = 1;
    iNumPages    = prndlg.PrinterSettings.MaximumPage;
    iPageNNumber = 1;

    predlg.ShowDialog();
}

protected override void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    prndlg.AllowSelection = textbox.SelectionLength > 0;

    if (prndlg.ShowDialog() == DialogResult.OK)
    {
        prndoc.DocumentName = Text;

        // Initialize some important fields.

        switch (prndlg.PrinterSettings.PrintRange)
        {
            case PrintRange.AllPages:
                strPrintText = textbox.Text;
                iStartPage   = 1;
                iNumPages    = prndlg.PrinterSettings.MaximumPage;
                break;

            case PrintRange.Selection:
                strPrintText = textbox.SelectedText;
                iStartPage   = 1;
                iNumPages    = prndlg.PrinterSettings.MaximumPage;
                break;

            case PrintRange.SomePages:
                strPrintText = textbox.Text;

                break;
        }
    }
}
void OnPrintPage((object obj, PrintPageEventArgs ppea)
{
    Graphics   grfx   = ppea.Graphics;
    Font       font   = textbox.Font;
    float      cyFont = font.GetHeight(grfx);
    StringFormat strfmt = new StringFormat();
    RectangleF  rectfFull, rectfText;
    int         iChars, iLines;

    // Calculate RectangleF for header and footer.
    if (grfx.VisibleClipBounds.X < 0)       // Print preview
        rectfFull = ppea.MarginBounds;
    else
        rectfFull = new RectangleF(
            ppea.MarginBounds.Left - (ppea.PageBBounds.Width -
            grfx.VisibleClipBounds.Width) / 2,
            ppea.MarginBounds.Top - (ppea.PageBBounds.Height -
            grfx.VisibleClipBounds.Height) / 2,
            ppea.MarginBounds.Width, ppea.MarginBounds.Height);

    // Calculate RectangleF for text.
    rectfText = RectangleF.Inflate(rectfFull, 0, -2 * cyFont);

    int iDisplayLines = (int)Math.Floor(rectfText.Height / cyFont);
    rectfText.Height = iDisplayLines * cyFont;

    // Set up StringFormat object for rectangular display of text.
    if (textbox.WordWrap)
    {
        strfmt.Trimming = StringTrimming.Word;
    }
}
else
{
    strfmt.Trimming = StringTrimming.EllipsisCharacter;
    strfmt.FormatFlags |= StringFormatFlags.NoWrap;
}

    // For printing only "some pagesv," get to the first page.
    while ((iPageNNumber < iStartPage) &&& (strPrintText.Length >
        0))
    {
        if (txtbox.WordWrap)
            grfx.MeasureString(strPrintText, font, rectfText.Size,
                                strfmt, out iChars, out iLines);
        else
            iChars = CharsInLines(strPrintText, iDisplayLines);

        strPrintText = strPrintText.Substring(iChars);
        iPageNNumber++;
    }

    // If we've prematurely run out of text, cancel print job.
    if (strPrintText.Length == 0)
    {
        ppea.Cancel = true;
        return;
    }

    // Display text for this page.
    grfx.DrawString(strPrintText, font, Brushes.Black,
                     rectfText, strfmt);

    // Get text for next page.
    if (txtbox.WordWrap)
        grfx.MeasureString(strPrintText, font, rectfText.Size,
                            strfmt, out iChars, out iLines);
    else
        iChars = CharsInLines(strPrintText, iDisplayLines);

    strPrintText = strPrintText.Substring(iChars);

    // Reset StringFormat display header and footer.
    strfmt = new StringFormat();

    // Display filename at top.
To make my program just a little better than Notepad, I decided to implement the two options in the 
PrintDialog dialog box that let you print a selection or a range of pages.

In this program, the PrintDialog object and all three print dialog boxes are defined as fields. These 
are initialized during the program's constructor. Also stored as fields are four variables that assist the 
program in printing a document.
The `MenuFilePrintOnClick` method is more extensive than anything we've seen so far, and it's primarily a result of implementing the options to print a selection or a range of pages as well as the entire document. For example, the `strPrintText` variable must equal the text to be printed. Normally, this variable is obtained from the `Text` property of the text box, but if a selection is to be printed, the variable must be obtained from `SelectedText`. To put it simply, the purpose of all this code is to set `iStartPage` (the first page to be printed) and `iNumPages` (the number of pages to be printed) to different values depending on which of the three printing options has been chosen.

The `MenuFilePreviewOnClick` method (which is invoked when the Print Preview menu item is clicked) sets `strPrintText`, `iStartPage`, `iNumPages`, and `iPageNumber` as if it were printing the whole document.

The `PrintPage` event handler begins by calculating a display rectangle. If printing normally, that involves the formula I showed earlier using `MarginBounds`, `PageBounds`, and `VisibleClipBounds`. However, if the `PrintPage` event handler is called as a result of a print preview, `VisibleClipBounds` indicates a drawing space larger than the bitmap, and the graphics origin is the upper left corner of the bitmap. In that case, the display rectangle is simply set to the `MarginBounds` property. When the word-wrap option is selected, the `PrintPage` event handler can rely on the word-wrapping abilities of `DrawString`. Notice how the method adjusts the calculated text display rectangle (`rectfText`) so that only an integral number of lines are displayed. That adjustment avoids clipping problems. Similarly, setting the `Trimming` property of the `StringFormat` object to `StringTrimming.Word` assures that the last word on the page isn't truncated. The `MeasureString` overload that returns the number of characters displayed is ideal for adjusting the `strPrintText` variable in preparation for the next page.

Non-word-wrapped text actually proved a little more difficult. The `OnPrintPage` method sets the `Trimming` property of the `StringFormat` object to `StringTrimming.EllipsisCharacter` and the `FormatFlags` property to `StringFormatFlags.NoWrap` to get the display of each page right. However, `MeasureString` insisted on returning the number of characters actually displayed, not the number that would have been displayed if each line weren't truncated. I was forced to write a little routine, `CharsInLines`, that searches for end-of-line characters and adjust `strPrintText` in that way.

Another approach to printing non-word-wrapped text would be to use the `Lines` property to obtain a `string` array containing all the individual lines of text. The `OnPrintPage` method could then simply call `DrawString` for each line. However, this approach would have worked only for printing the whole document or a range of pages. The `TextBoxBase` class doesn't have a property that returns the `selected` text broken down into lines.

The Print Preview dialog box contains a button to print the document before returning to the application. To handle that eventuality, when `OnPrintPage` has reached the last page, it also reinitializes the `strPrintText`, `iStartPage`, `iNumPages`, and `iPageNumber` fields in preparation for the regular print job.
Chapter 22: Tree View and List View

Overview

I take it you're familiar with the Microsoft Windows Explorer program. The client area of Windows Explorer is dominated by two large and sophisticated controls. The tree-view control on the left displays a hierarchical list of the user's disk drives and directories. The list-view control on the right displays the subdirectories and files in the selected directory in one of four formats: a simple list, a table with multiple columns, names with small icons, or names with large icons.

In this chapter, I'll discuss the Windows Forms implementation of the tree-view and list-view controls. The sophistication and versatility of these controls makes an exhaustive discussion impossible. But I'll certainly cover them in enough detail to get you started.

Before we begin, however, I'd like to direct your attention to a third control in Windows Explorer: that thin vertical bar that looks like a sizing border but appears between the tree-view and list-view controls. You move it sideways with the mouse to adjust how the client area of Windows Explorer is divided between the two controls. As one control gets larger, the other gets smaller. That's called a splitter control.

Splitsville

You've probably seen splitters in other applications besides Windows Explorer. In Microsoft Word, for example, you can choose Split from the Window menu and divide your document into two regions with a horizontal splitter. This feature allows you to work on one section of a document while keeping another section in view. Again, the splitter adjusts the relative sizes of the two views. By default, Microsoft Visual Studio .NET uses splitters to separate its client area into four areas.

Splitters are also used to display Web pages that make use of frames, a feature introduced in HTML 4.0. The HTML 4.0 Specification (Section 16.1) has a good explanation of the rationale behind frame-based architecture:

HTML frames allow authors to present documents in multiple views, which may be independent windows or subwindows. Multiple views offer designers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, a second a navigation menu, and a third the main document that can be scrolled through or replaced by navigating in the second frame.

Although the size of these frames can be fixed, by default they are adjustable with splitters. Both horizontal and vertical splitters often appear on the same Web page.

The Windows Forms splitter control is implemented in the Splitter class, which is based on Control. Once you've correctly created and positioned a splitter, you can generally ignore it. Only rarely will you need to process splitter events. A splitter affects the size of other controls in much the same way as a sizing border. If your controls react well to being resized, they will adapt just fine to being resized with splitters.

A splitter is always associated with a target control, which is the control that the splitter directly alters the size of, though other controls can also be affected by the splitter's movement. A splitter is attached to a target control through the mechanism of docking, which I introduced in Chapter 12. As you'll recall, the Control class implements a property named Dock that can be assigned one of the following members of the DockStyle enumeration:

<table>
<thead>
<tr>
<th>DockStyle Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Top</td>
</tr>
<tr>
<td>Bottom</td>
</tr>
<tr>
<td>Left</td>
</tr>
</tbody>
</table>
DockStyle Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>4</td>
</tr>
<tr>
<td>Fill</td>
<td>5</td>
</tr>
</tbody>
</table>

The default is DockStyle.None. When you specify one of the next four members in the table, the control is positioned flush against that side of its parent and extended to the two adjacent sides. For example, if control is an instance of any class descended from Control and you set the Dock property as

```csharp
control.Dock = DockStyle.Left;
```

then control will be moved to the far left side of its parent and will be resized to occupy the full space between the top and the bottom of the parent. Whenever the parent is resized, control will also be resized accordingly. The Control class performs this magic during the OnResize method. For that reason (and others), whenever you override OnResize, you should call the base class version of the method.

The DockStyle.Fill option causes the control to fill up the surface of its parent. I used DockStyle.Fill in the AnalogClock program in Chapter 10 and in the various Notepad clone programs beginning in Chapter 18.

What happens when you dock two or more controls against the same side? Or you use DockStyle.Fill with two or more controls? Or you use DockStyle.Fill with one control and one of the other nondefault DockStyle members with another control? In all these cases, the behavior is determined by the z-order of the controls.

I discussed z-order in Chapter 12, but here's a brief review. As you add controls to a parent, they are assigned a child index beginning with 0. A control with a child index of 0 is said to be at the top of the z-order. When controls overlap, the child control on the top of the z-order appears on top of the other controls and receives mouse events when the cursor passes over the control. The child control with the highest child index is said to be at the bottom of the z-order.

Programs can reorder controls by calling the BringToFront and SendToBack methods implemented by Control or by calling the SetChildIndex method implemented by Control.ControlCollection. It's usually easiest just to create the controls in the desired order in the first place.

So, what happens when two or more child controls have a Dock property equal to DockStyle.Fill? The child control at the top of the z-order—that is, the child control added to the parent earliest and therefore having the lowest child index—appears on top of the others. For example, consider the following code, which could appear in the constructor of a class derived from Form:

```csharp
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock = DockStyle.Fill;

Panel panel2 = new Panel();
panel2.Parent = this;
panel2.Dock = DockStyle.Fill;
```

The control named panel1 will be visible; panel2 will not. The general rule is that only one child control should have a Dock property of DockStyle.Fill. (I violate this rule in the ImageDirectory program later in this chapter. That program has two controls with DockStyle.Fill but at any time only one of them has a Visible property of true.)

It's possible to add other controls to the parent before the control that has a Dock property of DockStyle.Fill, for example,

```csharp
Button btn = new Button();
btn.Parent = this;
```
Panel panel = new Panel();
panel.Parent = this;
panel.Dock = DockStyle.Fill;

The Button control is at the top of the z-order and appears on top of the Panel control. The Panel control fills the client area of the parent. It might look as if the Button control is a child of the Panel control, but it's not. The controls are simply overlapping, and the one at the top of the z-order gets priority of visibility.

The following case is similar:
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock = DockStyle.Left;

Panel panel2 = new Panel();
panel2.Parent = this;
panel2.Dock = DockStyle.Fill;

Notice that the first Panel control has a Dock property of DockStyle.Left. It's on the top of the z-order, so it gets priority. The second panel will still fill the client area of its parent, but the left part will be obscured by panel1. That's probably not a desirable situation.

This case is much more useful and (with the addition of a splitter control) quite common:
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock = DockStyle.Fill;

Panel panel2 = new Panel();
panel2.Parent = this;
panel2.Dock = DockStyle.Left;

All I've done here is switch around the two Dock properties. Now both panels are fully visible. The first panel appears at the right side of the client area and the second panel appears at the left. As you make the client area wider and narrower, panel1 at the right changes size because that's the one with DockStyle.Fill; panel2 on the left side of the client area doesn't change size in that case.

I've already demonstrated this technique in the SimpleStatusBarWithPanel program toward the beginning of Chapter 20. That program begins by creating a Panel with a Dock property of DockStyle.Fill and then the StatusBar control, which has a default setting of DockStyle.Bottom.

If you change the last line of the preceding code to be
panel2.Dock = DockStyle.Right;

then panel1 appears at the left of the client area and panel2 at the right. As you change the size of the client area, panel1 (now at the left) changes size accordingly; panel2 remains the same size. A splitter control between these two panels would be ideal.

What happens when you use the same Dock property with two controls? Here's an example:
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock = DockStyle.Left;

Panel panel2 = new Panel();
As the second control is added to the parent, the control at the top of the z-order (panel1) gets pushed toward the center of the client area. The control at the bottom of the z-order is positioned at the left edge of the parent.

Here's an example of two controls that don't have the same Dock property but do have Dock properties that seem to conflict:

Panel panel1  = new Panel();
panel1.Parent = this;
panel1.Dock   = DockStyle.Left;

Panel panel2  = new Panel();
panel2.Parent = this;
panel2.Dock   = DockStyle.Top;

The first panel has a Dock property of DockStyle.Left, and the second has DockStyle.Top. Initially, the first panel is positioned to hug the left of the client area and extend from the top of the parent to the bottom. The second panel essentially pushes the first panel down. The second panel is positioned at the top of the client area and extends to the client's full width. The first panel ends up below the second panel. Both panels are fully visible.

And now we're ready to add splitters to the mix. To keep the discussion general, I'll continue to use Panel controls in these examples. These panels could, of course, contain other controls, and they could also be scrollable.

It's important to know what happens when multiple controls are docked against the same edge of a parent because—if you're not careful—you could very easily end up with splitters at the edge of the window, where they do no good. The target control of a splitter is the control that's docked to the same edge as the splitter but with a lower z-order. For this reason, you generally create the splitter before the target control.

Most commonly, splitter controls are placed between two controls so that moving the splitter causes one control to increase in size and the other control to get smaller. However, you can use splitters in an even simpler way to change the size of a single control. Here's a program that uses a splitter to resize a single panel control.

OnePanelWithSplitter.cs

//---------------------------------------------------
// OnePanelWithSplitter.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class OnePanelWithSplitter: Form
{
    public static void Main()
    {
        Application.Run(new OnePanelWithSplitter());
    }
    public OnePanelWithSplitter()
    {
        Text = "One Panel with Splitter";
    }
Both the splitter and the panel have a `Dock` property of `DockStyle.Left`. But remember that controls at the bottom of the z-order are closest to the docking edge. That's why the splitter is created first. The panel essentially pushes the splitter away from the left edge of the parent, so the splitter ends up on the right edge of the panel.

Normally, a panel is the same color as its parent, so I've deliberately changed the `BackColor` property of the panel control so that you won't have to rely on faith that the splitter control is actually changing its size. Throughout this section, I'll be using colors to indicate positioning: in this case, `lime` stands for `left`. I've also installed handlers for the panel's `Resize` and `Paint` events, so you're additionally assured that you're seeing the entire panel control. Here's what the program looks like:
You can’t actually see the splitter in this example because its `BackColor` property is the same as its parent’s. But if you move the mouse to the right side of the panel, the cursor changes to small parallel vertical lines. You can then adjust the size of the panel. Splitters don’t have a keyboard interface. The cursor used by splitters is `Cursors.VSplit` for vertical splitters (like this one) or `Cursors.HSplit` for horizontal splitters.

Here’s a program that shows you how to make a splitter affect two panels that together fill the client area. The first panel has a `Dock` property of `DockStyle.Fill`, and both the splitter and the second panel get `DockStyle.Right` properties.

```csharp
// TwoPanelsWithSplitter.cs
//----------------------------------------------------
// TwoPanelsWithSplitter.cs © 2001 by Charles Petzold
//----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class TwoPanelsWithSplitter : Form
{
    public static void Main()
    {
        Application.Run(new TwoPanelsWithSplitter());
    }
    public TwoPanelsWithSplitter()
    {
        Text = "Two Panels with Splitter";

        Panel panel1 = new Panel();
        panel1.Parent = this;
        panel1.Dock = DockStyle.Fill;
        panel1.BackColor = Color.Lime;
        panel1.Resize += new EventHandler(PanelOnResize);
        panel1.Paint += new PaintEventHandler(PanelOnPaint);
    }
}
```
Splitter split = new Splitter();
split.Parent = this;
split.Dock = DockStyle.Right;

Panel panel2 = new Panel();
panel2.Parent = this;
panel2.Dock = DockStyle.Right;
panel2.BackColor = Color.Red;
panel2.Resize += new EventHandler(PanelOnResize);
panel2.Paint += new PaintEventHandler(PanelOnPaint);
}

void PanelOnResize(object obj, EventArgs ea)
{
    ((Panel) obj).Invalidate();
}

void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;

    grfx.DrawEllipse(Pens.Black, 0, 0, 
                      panel.Width - 1, panel.Height - 1);
}

As you’re experimenting with this program, you’ll see the lime panel on the left and the red panel on 
the right. (Notice the mnemonics.) The splitter is more visible now because it’s gray. Here's how the 
program starts up:

The target of the splitter is the panel on the right because they both have Dock properties of 
DockStyle.Right, but the splitter actually affects the size of both panels. But what happens when you 
make the parent form wider or narrower? The first panel—the one on the left—changes size because
that's the one with DockStyle.Fill. This behavior has an additional implication, however: if you make
the client area too narrow, the panel on the left will disappear from view and the panel on the right
will be clipped as well.

When you create a program with two controls (such as panels) separated by a splitter, give some
thought to which control should be affected by changes in the parent's size. Create that one first with
DockStyle.Fill.

Here's a program that uses that rule to create a form with three panels. The center one (colored cyan
for center) has the DockStyle.Fill property, so that's the one that changes size as you change the
client area size.

**SplitThreeAcross.cs**

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitThreeAcross: Form
{
    public static void Main()
    {
        Application.Run(new SplitThreeAcross());
    }
    public SplitThreeAcross()
    {
        Text = "Split Three Across";

        Panel panel1 = new Panel();
        panel1.Parent = this;
        panel1.Dock = DockStyle.Fill;
        panel1.BackColor = Color.Cyan;
        panel1.Resize += new EventHandler(PanelOnResize);
        panel1.Paint += new PaintEventHandler(PanelOnPaint);

        Splitter split1 = new Splitter();
        split1.Parent = this;
        split1.Dock = DockStyle.Left;

        Panel panel2 = new Panel();
        panel2.Parent = this;
        panel2.Dock = DockStyle.Left;
        panel2.BackColor = Color.Lime;
        panel2.Resize += new EventHandler(PanelOnResize);
        panel2.Paint += new PaintEventHandler(PanelOnPaint);

        Splitter split2 = new Splitter();
```
split2.Parent = this;
split2.Dock = DockStyle.Right;

Panel panel3 = new Panel();
panel3.Parent = this;
panel3.Dock = DockStyle.Right;
panel3.BackColor = Color.Red;
panel3.Resize += new EventHandler(PanelOnResize);
panel3.Paint += new PaintEventHandler(PanelOnPaint);

panel1.Width =
panel2.Width =
panel3.Width = ClientSize.Width / 3;
}
void PanelOnResize(object obj, EventArgs ea)
{
    ((Panel) obj).Invalidate();
}
void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;

    grfx.DrawEllipse(Pens.Black, 0, 0,
                      panel.Width - 1, panel.Height - 1);
}

Again, because the center panel changes size when you resize the form, the center panel can also
disappear from view if you make the client area too narrow. As you make the client area even
narrower, the right panel slides under the left panel because the left panel is at the top of the z-order.
To make sure that all panels are visible when the program starts up, at the end of the constructor, I
give them each a width equal to 1/3 the client area.
In an arrangement like this, you don’t need to make the center panel the one that changes size with the client area. The requirements of a particular application will determine which approach seems most natural.

Perhaps a better approach when designing dual-splitter forms is to begin with a single splitter controlling the size of two controls and then make two more controls and a splitter as children of one of those existing controls. You use this approach when you’re mixing horizontal and vertical splitters in a form that resembles an HTML frame, as in the following program.

```csharp
SplitThreeFrames.cs
// SplitThreeFrames.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitThreeFrames: Form
{
    public static void Main()
    {
        Application.Run(new SplitThreeFrames());
    }
    public SplitThreeFrames()
    {
        Text = "Split Three Frames";
        Panel panel = new Panel();
        panel.Parent = this;
        panel.Dock = DockStyle.Fill;
    
        Splitter split1 = new Splitter();
        split1.Parent = this;
        split1.Dock = DockStyle.Left;
    }
}
```
Panel panel1 = new Panel();
panel1.Parent = this;
panel1.Dock = DockStyle.Left;
panel1.BackColor = Color.Lime;
panel1.Resize += new EventHandler(PanelOnResize);
panel1.Paint += new PaintEventHandler(PanelOnPaint);

Panel panel2 = new Panel();
panel2.Parent = panel;
panel2.Dock = DockStyle.Fill;
panel2.BackColor = Color.Blue;
panel2.Resize += new EventHandler(PanelOnResize);
panel2.Paint += new PaintEventHandler(PanelOnPaint);

Splitter split2 = new Splitter();
split2.Parent = panel;
split2.Dock = DockStyle.Top;

Panel panel3 = new Panel();
panel3.Parent = panel;
panel3.Dock = DockStyle.Top;
panel3.BackColor = Color.Tan;
panel3.Resize += new EventHandler(PanelOnResize);
panel3.Paint += new PaintEventHandler(PanelOnPaint);

panel1.Width = ClientSize.Width / 3;
panel3.Height = ClientSize.Height / 3;
}

void PanelOnResize(object obj, EventArgs ea)
{
    ((Panel) obj).Invalidate();
}

void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;

    grfx.DrawEllipse(Pens.Black, 0, 0,
                     panel.Width - 1, panel.Height - 1);
}

The SplitThreeFrames program begins by creating two panels (on the right and the left) with a vertical splitter between them. The first panel (named simply panel) gets the DockStyle.Fill property. I don't give it a color because this panel will be a parent to other panels that will entirely cover its
surface. The second panel (named panel1 and colored lime) is created with DockStyle.Left. The result consists of two panels with a vertical splitter between them.

But wait, there's more! The first panel (named panel) becomes a parent to two more panels and a splitter. The first child (panel2) of that panel gets the DockStyle.Fill property and a color of blue (for bottom). The program then creates another splitter and a panel (panel3) colored tan (for top) with DockStyle.Top properties.

The constructor concludes by setting the initial size of the left and top panels.

As you change the size of the client area, the panel on the bottom right changes size, but that's only because it has a DockStyle.Fill property and is a child of another panel that has a DockStyle.Fill property.

The only properties of the Splitter class I've been using so far are Parent and Dock. The following properties of the Splitter class (with a couple inherited from Control) are probably the most useful:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>SplitPosition</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MinSize</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>MinExtra</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get/set</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>BorderStyle</td>
<td>get/set</td>
</tr>
<tr>
<td>Color</td>
<td>BackColor</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The SplitPosition property indicates the width of the target control, if the splitter is vertical, or the height of the target control, if the splitter is horizontal. If the splitter is not yet bound to a target control, however, the property will equal −1. When you create splitters in a constructor (such as we've been doing), the splitters aren't assigned target controls until after the constructor concludes. For that reason, don't use SplitPosition in the constructor. If you need to initialize the relative sizes of controls that use splitters, do it by sizing the controls, as the SplitThreeAcross and SplitThreeFrames programs demonstrate.

The MinSize property indicates the minimum width (or height) to which you can resize the target control using the splitter. The MinExtra property indicates the minimum width (or height) of the control on the other side of the splitter. By default, these properties are set to 25 pixels. (You can easily see the effect of these properties by using any of the programs shown so far in this chapter.)
wouldn't recommend that you set these properties to 0 because a user might get confused if a control shrinks down to nothing. But you can set them very low if you want to give the user the opportunity to move the control almost completely out of the way.

The *Width* property is the width of the splitter (by default, 3 pixels). By default, the *BorderStyle* property of the splitter is *BorderStyle.None*, which makes the splitter a simple strip of unadorned *BackColor*. *BorderStyle.Fixed3D* is the same as *BorderStyle.None*. *BorderStyle.FixedSingle* makes the splitter stand out more by coloring the outer edges black. The *ForeColor* property has no effect on splitters.

The *Splitter* class adds two events to those implemented in *Control*:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SplitterMoving</td>
<td>OnSplitterMoving</td>
<td>SplitterEventHandler</td>
<td>SplitterEventArgs</td>
</tr>
<tr>
<td>SplitterMoved</td>
<td>OnSplitterMoved</td>
<td>SplitterEventHandler</td>
<td>SplitterEventArgs</td>
</tr>
</tbody>
</table>

Both events are delivered with an object of type *SplitterEventArgs*, which has the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>X</td>
<td>get</td>
<td>Mouse cursor position</td>
</tr>
<tr>
<td>int</td>
<td>Y</td>
<td>get</td>
<td>Mouse cursor position</td>
</tr>
<tr>
<td>int</td>
<td>SplitX</td>
<td>get/set</td>
<td>Splitter position</td>
</tr>
<tr>
<td>int</td>
<td>SplitY</td>
<td>get/set</td>
<td>Splitter position</td>
</tr>
</tbody>
</table>

All positions are relative to the parent window of the splitter. The *SplitX* and *SplitY* properties indicate the position of the upper left corner of the splitter relative to the client area. For vertical splitters, *SplitY* equals 0, and for horizontal splitters, *SplitX* equals 0.

Here's a program that creates two panels with a splitter. Initially, each panel occupies about half the client area, which, as you know, you can change with the splitter. However, when you resize the form, the two panels change size proportionally.

**SplitTwoProportional.cs**

```csharp
//---------------------------------------------------
// SplitTwoProportional.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitTwoProportional: Form
{
    Panel panel2;
    float fProportion = 0.5f;

    public static void Main()
    {
        Application.Run(new SplitTwoProportional());
    }
}
public SplitTwoProportional()
{
    Text = "Split Two Proportional";

    Panel panel1 = new Panel();
    panel1.Parent = this;
    panel1.Dock = DockStyle.Fill;
    panel1.BackColor = Color.Red;
    panel1.Resize += new EventHandler(PanelOnResize);
    panel1.Paint += new PaintEventHandler(PanelOnPaint);

    Splitter split = new Splitter();
    split.Parent = this;
    split.Dock = DockStyle.Left;
    split.SplitterMoving += new SplitterEventHandler(SplitterOnMoving);

    panel2 = new Panel();
    panel2.Parent = this;
    panel2.Dock = DockStyle.Left;
    panel2.BackColor = Color.Lime;
    panel2.Resize += new EventHandler(PanelOnResize);
    panel2.Paint += new PaintEventHandler(PanelOnPaint);

    OnResize(EventArgs.Empty);
}

protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);
    panel2.Width = (int) (fProportion * ClientSize.Width);
}

void SplitterOnMoving(object obj, SplitterEventArgs sea)
{
    fProportion = (float) sea.SplitX / ClientSize.Width;
}

void PanelOnResize(object obj, EventArgs ea)
{
    ((Panel) obj).Invalidate();
}

void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;

    grfx.DrawEllipse(Pens.Black, 0, 0,
                     panel.Width - 1, panel.Height - 1);
This program retains an instance variable named \textit{fProportion} equal to the ratio of the panel on the left to the width of the client area. It's initialized to 0.5 and altered whenever the user moves the splitter. That's the SplitterOnMoving event. When the user resizes the program's client area, the OnResize method sets the size of the left panel based on the \textit{fProportion} value and the new width of the client. (I used the SplitterMoving event rather than SplitterMoved because the latter wasn't working as documented at the time I wrote the program.)

Now that we know what kind of control goes between a tree view and a list view, we're ready to look at these two controls in detail.

\section*{Tree Views and Tree Nodes}

The \textit{TreeView} control is most commonly used to display a list of disk drives and directories like the one at the left side of Windows Explorer. However, you can use \textit{TreeView} controls for displaying any hierarchical information. Visual Studio .NET uses a tree view for displaying projects and files, and the Microsoft Document Explorer (the program that displays the .NET programming documentation) also uses a tree view for displaying namespaces, classes, members, and so forth.

The bulk of the tree view implementation in Windows Forms consists of the \textit{TreeView}, \textit{TreeNode}, and \textit{TreeNodeCollection} classes. An object of type \textit{TreeNode} is a single entry in the tree view. A \textit{TreeNode} object is associated with a string and an optional image. In Windows Explorer, the string is a drive or directory name, and the images resemble disk drives and folders.

The \textit{TreeNode} class contains a property named \textit{Nodes} that is a collection of other \textit{TreeNode} objects:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{TreeNode Properties (selection)} &  & \\
\hline
\textbf{Type} & \textbf{Property} & \textbf{Accessibility} \\
\hline
TreeNodeCollection & Nodes & get \\
\hline
\end{tabular}
\end{table}

The \textit{Nodes} property contains all the subnodes (or child nodes) of the node. \textit{TreeNodeCollection} is a familiar sort of class. It implements the \textit{IList}, \textit{ICollection}, and \textit{IEnumerable} interfaces, and it lets you index the collection like an array:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{TreeNodeCollection Properties} &  & \\
\hline
\textbf{Type} & \textbf{Property} & \textbf{Accessibility} \\
\hline
TreeNode & [] & get/set \\
int & Count & get \\
bool & IsReadOnly & get \\
\hline
\end{tabular}
\end{table}

The customary way to add child nodes to an existing node is through the \textit{Add} and \textit{AddRange} methods of \textit{TreeNodeCollection}:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{TreeNodeCollection Methods (selection)} &  & \\
\hline
TreeNode & Add(string strNode) & \\
int & Add(TreeNode node) & \\
void & AddRange(TreeNode[] anode) & \\
\hline
\end{tabular}
\end{table}

\textit{TreeNodeCollection} also contains familiar methods named \textit{Insert}, \textit{Remove}, \textit{Clear}, and others. All the nodes in the same \textit{TreeNodeCollection} object are sometimes referred to collectively as \textit{siblings}. The \textit{TreeNode} object to which the collection belongs is the parent.

I haven't yet said anything about \textit{TreeView}. Basically, \textit{TreeView} is a collection of top-level (or root) \textit{TreeNode} objects. Like \textit{TreeNode}, \textit{TreeView} contains a \textit{Nodes} property:
This **Nodes** property is a collection of all the root **TreeNode** objects.

Conceptually, **TreeView** is similar to **MainMenu** or **ContextMenu**, while **TreeNode** is similar to **MenuItem**. (See Chapter 14 for information on menu objects.) That is, just as **MainMenu** or **ContextMenu** is a collection of nested **MenuItem** objects, **TreeView** is a collection of nested **TreeNode** items. However, as you'll recall, all three of these menu-related classes are derived from the **Menu** class. **TreeView** and **TreeNode** are not related in that way: **TreeView** derives from **Control**, while **TreeNode** derives from **MarshalByRefObject**.

And with this information, we're ready to construct our first tree. Here's a program that has the beginnings of an animal/mineral/vegetable hierarchy.

**SimpleTreeView.cs**
```
// SimpleTreeView.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Windows.Forms;

class SimpleTreeView: Form
{
    public static void Main()
    {
        Application.Run(new SimpleTreeView());
    }
    public SimpleTreeView()
    {
        Text = "Simple Tree View";
        TreeView tree = new TreeView();
        tree.Parent = this;
        tree.Dock = DockStyle.Fill;
        tree.Nodes.Add("Animal");
        tree.Nodes[0].Nodes.Add("Dog");
        tree.Nodes[0].Nodes[0].Nodes.Add("Poodle");
        tree.Nodes[0].Nodes[0].Nodes.Add("Irish Setter");
        tree.Nodes[0].Nodes[0].Nodes.Add("German Shepherd");
        tree.Nodes[0].Nodes.Add("Cat");
        tree.Nodes[0].Nodes[1].Nodes.Add("Calico");
        tree.Nodes[0].Nodes[1].Nodes.Add("Siamese");
        tree.Nodes[0].Nodes.Add("Primate");
        tree.Nodes[0].Nodes[2].Nodes.Add("Chimpanzee");
        tree.Nodes[0].Nodes[2].Nodes.Add("Ape");
    }
```
The constructor creates the entire `TreeView` control through the use of 20 calls to the `Add` method in `TreeNodeCollection`. Three of these calls involve the `Nodes` property of the `TreeView` object and hence create top-level nodes:

```csharp
    tree.Nodes.Add("Animal");
    :
    :
    tree.Nodes.Add("Vegetable");
```

Although the program doesn't refer explicitly to any `TreeNode` objects, they are certainly there. Each of the `Add` methods creates a `TreeNode` object. The second `Add` call is this one:

```csharp
    tree.Nodes[0].Nodes.Add("Dog");
```

The first part of that statement (`tree.Nodes[0]`) refers to the first `TreeNode` object in the `TreeView` object's collection, that is, "Animal". The second `Nodes` property is the collection of child `TreeNode` objects of "Animal", to which the node "Dog" is added. Similarly, the following statement adds a child node under "Dog":

```csharp
    tree.Nodes[0].Nodes[0].Nodes.Add("Poodle");
```

Here's the program with some of the nodes expanded:

You'll want to experiment with this program a bit to get the hang of the default user interface implemented in `TreeView` controls. Using the up and down cursor-movement keys, you can change the selected item (indicated by reverse video). If the item has a plus sign to its left, you can use the
right arrow key to view the child nodes. The left arrow key has two functions. When the selected
node doesn't have any children, it causes the selection to jump to the parent node. When the
selected node is already expanded (that is, has a minus sign next to it), the left arrow key collapses
the node. Of course, you can also click with the mouse on the pluses and minuses to expand and
collapse nodes.

Scroll bars are displayed by default when they are needed. If you don't want them, you can set the
Scrollable property of TreeView to false.

Although the SimpleTreeView program creates the entire TreeView control right in its constructor, a
program must often modify the contents of a TreeView control later on, at runtime. You can cause
items to be sorted by setting the following property to true:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Sorted</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

Regardless of whether or not the items are sorted, to prevent performance problems when modifying
a TreeView control, call the following methods before and after any sequence of statements that
affect multiple nodes:

```
void BeginUpdate()
void EndUpdate()
```

**Images in Tree Views**

As you can see from the SimpleTreeView program, you get the plus signs and minus signs for free.
However, you can suppress them if you want to. The following properties are all true by default:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>ShowPlusMinus</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>ShowLines</td>
<td>get/set</td>
</tr>
<tr>
<td>bool</td>
<td>ShowRootLines</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```

If ShowLines is false, all the lines that normally connect the nodes are not displayed. If
ShowRootLines is false, the root items are displayed without any lines or pluses and minuses. The
other items are displayed normally.

TreeView controls often display little pictures to the left of each node. Often these images change
when the node is expanded. For example, Windows Explorer shows a closed folder for a collapsed
directory node and an open folder for an expanded directory node.

Images in a TreeView control are based on a single ImageList object that applies to the entire
control:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageList</td>
<td>ImageList</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Imagelndex</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>SelectedImageIndex</td>
<td>get/set</td>
</tr>
</tbody>
</table>
```
The `ImageIndex` property indicates the default image displayed for a node that is not selected. `SelectedImageIndex` indicates the default image for a selected node.

More commonly, you'll want to specify indices for each `TreeNode` object:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>ImageIndex</code></td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td><code>SelectedImageIndex</code></td>
<td>get/set</td>
</tr>
</tbody>
</table>

However, `TreeNode` doesn't have its own `ImageList` property. The indices refer to the `ImageList` property in the `TreeView` control to which the `TreeNode` object belongs.

You can also specify these image indices when you create a `TreeNode` object. Here's a complete list of the `TreeNode` constructors:

### TreeNode Constructors

- `TreeNode()`
- `TreeNode(string strNode)`
- `TreeNode(string strNode, TreeNode[] anodes)`
- `TreeNode(string strNode, int indexImage, int indexImageSelected)`
- `TreeNode(string strNode, int indexImage, int indexImageSelected, TreeNode[] anodes)`

In the SimpleTreeView program, I used the `Add` method of `TreeNodeCollection` that has a string argument. Another version of `Add` has a `TreeNode` argument. Thus, it's possible to create a `TreeNode` object first and then add it to a `TreeNodeCollection` object. Two of the `TreeNode` constructors let you specify arrays of child `TreeNode` objects. These constructors let you build up a `TreeView` hierarchy from the lowest descendents up to the root.

### Tree View Events

The `TreeNode` class doesn't define any events on its own. However, `TreeView` implements 11 events in addition to the ones it inherits from `Control`. Here are the 6 crucial ones:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeforeExpand</td>
<td><code>OnBeforeExpand</code></td>
<td><code>TreeViewCancel-EventHandler</code></td>
<td><code>TreeViewCancel-EventArgs</code></td>
</tr>
<tr>
<td>BeforeCollapse</td>
<td><code>OnBeforeCollapse</code></td>
<td><code>TreeViewCancel-EventHandler</code></td>
<td><code>TreeViewCancel-EventArgs</code></td>
</tr>
<tr>
<td>BeforeSelect</td>
<td><code>OnBeforeSelect</code></td>
<td><code>TreeViewCancel-EventHandler</code></td>
<td><code>TreeViewCancel-EventArgs</code></td>
</tr>
<tr>
<td>AfterExpand</td>
<td><code>OnAfterExpand</code></td>
<td><code>TreeViewEventHandler</code></td>
<td><code>TreeViewEventArgs</code></td>
</tr>
<tr>
<td>AfterCollapse</td>
<td><code>OnAfterCollapse</code></td>
<td><code>TreeViewEventHandler</code></td>
<td><code>TreeViewEventArgs</code></td>
</tr>
<tr>
<td>AfterSelect</td>
<td><code>OnAfterSelect</code></td>
<td><code>TreeViewEventHandler</code></td>
<td><code>TreeViewEventArgs</code></td>
</tr>
</tbody>
</table>

These events occur when the user (or the program) expands, collapses, or selects a node. As you can see, the events come in pairs. The events that begin with the word `Before` occur before the
TreeView carries out the operation. The `TreeViewCancelEventArgs` object that accompanies these events has the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode</td>
<td>Node</td>
<td>get</td>
</tr>
<tr>
<td>TreeViewAction</td>
<td>Action</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>Cancel</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `Node` property indicates the `TreeNode` object that the user is attempting to expand, collapse, or select. The `Action` property is a member of the following enumeration:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>0</td>
</tr>
<tr>
<td>ByKeyboard</td>
<td>1</td>
</tr>
<tr>
<td>ByMouse</td>
<td>2</td>
</tr>
<tr>
<td>Collapse</td>
<td>3</td>
</tr>
<tr>
<td>Expand</td>
<td>4</td>
</tr>
</tbody>
</table>

If, for one reason or another, the program decides that it can't let the operation proceed, the event handler can set the `Cancel` property (inherited from `CancelEventArgs`) to `true`.

Otherwise, the expand, collapse, or select will be carried out by the `TreeView` control and the events beginning with the word `After` will occur. The accompanying `TreeViewEventArgs` object has the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode</td>
<td>Node</td>
<td>get</td>
</tr>
<tr>
<td>TreeViewAction</td>
<td>Action</td>
<td>get</td>
</tr>
</tbody>
</table>

I won't be demonstrating any of the other events implemented by `TreeView`. The `BeforeLabelEdit` and `AfterLabelEdit` events occur only if the `LabelEdit` property is set to `true`. This facility lets users edit the text of a tree node. The `BeforeCheck` and `AfterCheck` events occur only if the `Checkboxes` property is `true`, indicating that check boxes the user can check are placed next to the nodes on the tree. `TreeNode` has a property named `Checked` that indicates whether the node is checked. The `ItemDrag` event occurs when something is dragged to the `TreeView` control.

Keep in mind that `TreeView` inherits many methods, properties, and events from `Control`. For example, if you want to implement a context menu with items based on what node was right-clicked, you can install a handler for the `MouseDown` event and pass the mouse coordinates to the `GetNodeAt` method of `TreeView`.

**TreeNode Navigation**

When a `TreeView` event handler gets called, it usually must carry out some activity depending on the particular `TreeNode` object being expanded, collapsed, or selected. The nodes can be identified in several ways. Here are some useful basic properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeView</td>
<td>TreeView</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Index</td>
<td>get</td>
</tr>
</tbody>
</table>
The `TreeView` property indicates the `TreeView` control that the `TreeNode` object is part of. The `Index` is the index of the node in the collection of its siblings. The `Text` property is the text displayed by the node, obviously, and the `Tag` property allows the attachment of arbitrary information to the node for identification (or other) purposes.

The `TreeNode` class also includes several read-only properties that let a program navigate through the nodes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode</td>
<td>Parent</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>FirstNode</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>LastNode</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>NextNode</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>PrevNode</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>NextVisibleNode</td>
<td>get</td>
</tr>
<tr>
<td>TreeNode</td>
<td>PrevVisibleNode</td>
<td>get</td>
</tr>
</tbody>
</table>

The `Parent` property indicates the parent node. The `FirstNode` and `LastNode` properties refer to child nodes. (These are also available from the node's `Nodes` property.) The `NextNode` and `PrevNode` properties refer to sibling nodes. The `NextVisibleNode` and `PrevVisibleNode` properties could refer to siblings, children, or parents. These are the next (or previous) nodes that would be selected using the up and down arrow keys.

The `TreeNode` class has two properties that let a program determine whether a node is expanded, collapsed, or selected:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>IsExpanded</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IsSelected</td>
<td>get</td>
</tr>
</tbody>
</table>

A program can expand or collapse a `TreeNode` without any help from the user:

<table>
<thead>
<tr>
<th>TreeNode Methods (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Expand()</td>
</tr>
<tr>
<td>void ExpandAll()</td>
</tr>
<tr>
<td>void Collapse()</td>
</tr>
<tr>
<td>void Toggle()</td>
</tr>
</tbody>
</table>

The `ExpandAll` method expands all child nodes of the node for which the method is called. To expand or collapse the entire tree, you use these methods in `TreeView`.

<table>
<thead>
<tr>
<th>TreeView Methods (selection)</th>
</tr>
</thead>
</table>

```csharp
void ExpandAll()
```
void ExpandAll()
void CollapseAll()

You can use this property of `TreeView` to obtain or set the selected node:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeNode</td>
<td>SelectedNode</td>
<td>get/set</td>
</tr>
</tbody>
</table>

There is one remaining—and sometimes quite convenient—technique for identifying a particular node, available from this property of `TreeNode`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>FullPath</td>
<td>get</td>
</tr>
</tbody>
</table>

For any node, `FullPath` returns a text string that is formed by concatenating the text of the node with all its parent nodes going back to the root. The text strings are separated by the following character:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>PathSeparator</td>
<td>get/set</td>
</tr>
</tbody>
</table>

By default, `PathSeparator` is the backslash. In the SimpleTreeView program, the `FullPath` property for the "Siamese" node is "Animal\Cats\Siamese".

This `FullPath` property is ideal when you're working with a `TreeView` control that displays disks and directories, which (not coincidentally) is the next task we're going to tackle.

**The Directory Tree**

As we all know, displaying disks and directories is the most common job of a `TreeView` control. One might even expect the Windows Forms library to contain a class descended from `TreeView` that implements a standard directory tree. But even if there were, using it surely wouldn't be as much fun as making our own!

The following `DirectoryTreeView` class derives from `TreeView` and is used in the next two programs in this chapter and in another program (ExplorerLike) at the end of this chapter. The class makes use of some file I/O classes that I go over in more detail in Appendix A.

**DirectoryTreeView.cs**

```csharp
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class DirectoryTreeView: TreeView
{
    public DirectoryTreeView()
    {
```
// Make a little more room for long directory names.

Width *= 2;

// Get images for tree.

ImageList = new ImageList();
ImageList.Images.Add(new Bitmap(GetType(), "35FLOPPY.BMP"));
ImageList.Images.Add(new Bitmap(GetType(), "CLSDFOLD.BMP"));
ImageList.Images.Add(new Bitmap(GetType(), "OPENFOLD.BMP"));

// Construct tree.

RefreshTree();
}
public void RefreshTree()
{
    // Turn off visual updating and clear tree.

    BeginUpdate();
    Nodes.Clear();

    // Make disk drives the root nodes.

    string[] astrDrives = Directory.GetLogicalDrives();

    foreach (string str in astrDrives)
    {
        TreeNode tnDrive = new TreeNode(str, 0, 0);
        Nodes.Add(tnDrive);
        AddDirectories(tnDrive);

        if (str == "C:\")
            SelectedNode = tnDrive;
    }
    EndUpdate();
}
void AddDirectories(TreeNode tn)
{
    tn.Nodes.Clear();

    string strPath = tn.FullPath;
    DirectoryInfo dirinfo = new DirectoryInfo(strPath);
    DirectoryInfo[] adirinfo;
try {
    adirinfo = dirinfo.GetDirectories();
}
catch {
    return;
}

foreach (DirectoryInfo di in adirinfo) {
    TreeNode tnDir = new TreeNode(di.Name, 1, 2);
    tn.Nodes.Add(tnDir);

    // We could now fill up the whole tree with this statement:
    //    AddDirectories(tnDir);
    // But it would be too slow. Try it!
}

protected override void OnBeforeExpand(TreeViewCancelEventArgs tvcea) {
    base.OnBeforeExpand(tvcea);

    BeginUpdate();

    foreach (TreeNode tn in tvcea.Node.Nodes)
        AddDirectories(tn);

    EndUpdate();
}

DirectoryTreeView requires three small bitmaps that I copied from the collection provided with Visual Studio .NET. The directory, by default, is Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps\Outline\NoMask. Although I would have preferred displaying different images depending on the type of drive (floppy, hard disk, CD-ROM, and so forth), it's not possible using the Windows Forms classes to obtain the drive type. The construction of the ImageList object in the constructor assumes that the resource namespace is the empty string.
The `DirectoryTreeView` class implements one public method, named `RefreshTree`, that programs using the class can call to refresh the entire directory structure. (As you probably know, programs that use tree views displaying directories generally have a menu item named `Refresh`.) The constructor also calls `RefreshTree` to construct the tree.

`RefreshTree` obtains string representations of the system's disk drives by calling the static `Directory.GetLogicalDrives` method. This method returns an array of strings generally beginning with "A:\", "C:\", and so on. These strings become the root nodes. For each drive, `RefreshTree` calls `AddDirectories`.

`AddDirectories` has a `TreeNode` argument and is responsible for creating child nodes consisting of subdirectory names. The method uses the wonderful `FullPath` property of `TreeNode` to create a `DirectoryInfo` object. The `GetDirectories` method of `DirectoryInfo` then obtains an array of `DirectoryInfo` objects that are used to make child nodes.

It's possible that `DirectoryInfo` will raise an exception. This happens for a floppy disk drive if no diskette is present, for example, and even for some directories to which access is denied. For that reason, the method is called in a `try` block. Unfortunately, if a disk drive is empty (as is so often the case for drive A), `GetDirectories` also displays an annoying message box reporting the problem to the user before raising the exception. (Press Cancel or Continue to make the message box go away.) The message box is even displayed when console applications call `GetDirectories`! It's obviously a design flaw or a bug in `GetDirectories`, but until it's fixed, there's no way to prevent the message box from popping up. If it's really intolerable for your application, use the Win32 API functions `FindFirstFile` and `FindNextFile` instead.

I am well aware that every programmer faced with the job of constructing a directory tree immediately thinks `recursive function`. In fact, `AddDirectories` can indeed be called recursively to construct the entire directory tree. I've even included a statement (commented out, however) that calls `AddDirectories` recursively. You're welcome to remove the double slashes and see for yourself why I rejected this approach: it just takes too much time. It's much more efficient to call `AddDirectories` only when it's needed.

So why, you ask, does the `RefreshTree` method call `AddDirectories` at all? Initially, the tree needs to display only the disk drives. Calling `AddDirectories` for each disk drive seems unnecessary. However, disk drives that contain directories must be displayed with a plus sign that allows the user to expand the node. The only way to get the plus signs displayed is to add child nodes. So, even though only the disk drives are displayed initially, the subdirectories of the root of each drive are also added to the tree.

The `DirectoryTreeView` class also overrides the `OnBeforeExpand` method of `TreeView`. The first time this method will be called is when the user expands one of the disk drive nodes. However, `OnBeforeExpand` doesn't need to build the child nodes of the disk drive. Those already exist. Instead, the method needs to build child nodes for each of the newly displayed nodes—again, for the sole purpose of forcing `TreeView` to display a plus sign if the directory contains subdirectories.

Here's a program that makes use of `DirectoryTreeView` in a very simple way. This program creates a `Panel` control on the right, a `DirectoryTreeView` control on the left, and a `Splitter` in between. It installs an event handler for the `AfterSelect` event that `DirectoryTreeView` inherits from `TreeView` and displays a list of files in that directory in the panel.

```csharp
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class DirectoriesAndFiles: Form
```
{ }
panel.Invalidate();
}
void PanelOnPaint(object obj, PaintEventArgs pea)
{
    if (tnSelect == null)
        return;
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;
    DirectoryInfo dirinfo = new DirectoryInfo(tnSelect.FullPath);
    FileInfo[] afileinfo;
    Brush brush = new SolidBrush(panel.ForeColor);
    int y = 0;
    try
    {
        afileinfo = dirinfo.GetFiles();
    }
    catch
    {
        return;
    }
    foreach (FileInfo fileinfo in afileinfo)
    {
        grfx.DrawString(fileinfo.Name, Font, brush, 0, y);
        y += Font.Height;
    }
}
void MenuOnRefresh(object obj, EventArgs ea)
{
    dirtree.RefreshTree();
}

Because this is only a demonstration program, the list of files is only one column long, so the list might be truncated. Here’s a view of one of the subdirectories of my WINNT directory:
This program also has a View menu with one item: Refresh. The menu item rebuilds the directory tree by calling the RefreshTree method in the DirectoryTreeView class.

Displaying Images

In Chapter 23, I'll be delving into metafiles, which are binary collections of graphics drawing commands that describe an image. In preparation for that chapter, I wanted to look at some metafile clip art that I had. Generally, when I want to look at a directory full of images, I use a particular freeware program. The program displays a tree view on the left and thumbnails on the right. You click on a thumbnail to see the full-size image. But while the program works fine with many different bitmap formats, it doesn't read metafiles at all. Recent versions of Windows Explorer display thumbnails of bitmaps and metafiles, but Windows Explorer requires an external program for displaying the full-size images.

In Windows Forms, both the Bitmap and Metafile classes are descended from Image. Metafiles can be read from the disk using the static Image.FromFile method just as easily as bitmaps, and metafiles can also be displayed as easily with DrawImage. In a Windows Forms program that loads and displays bitmaps, metafile support is free.

Let's set some simple goals. A program named ImageDirectory will display a TreeView control on the left displaying directories. On the right, the program will display a collection of thumbnails showing all the image files (bitmaps and metafiles) in the selected directory. Click on a thumbnail to see the image enlarged to the size of the form.

We've already written a good chunk of this program. That's the DirectoryTreeView control. The other half of the program's client area will consist of a Panel control. Each thumbnail is a Button control that displays the image scaled down to the size of a button. Here's an ImagePanel control that descends from Panel to do this job.

ImagePanel.cs

// --------------------------------------------------------------
// ImagePanel.cs © 2001 by Charles Petzold
// --------------------------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class ImagePanel : Panel
{
    const int cxButton = 100, cyButton = 100;            // Image button size
    Button btnClicked;
    ToolTip tooltip = new ToolTip();
    Timer timer = new Timer();

    // Fields for Timer Tick event
    string[] astrFileNames;
    int i, x, y;                                          // Public event
    public event EventHandler ImageClicked;              // Constructor

    public ImagePanel()
AutoScroll = true;
timer.Interval = 1;
timer.Tick += new EventHandler(TimerOnTick);
}

// Public properties
public Control ClickedControl
{
    get { return btnClicked; }
}

dublic Image ClickedImage
{
    get
    {
        try
        {
            return Image.FromFile((string) btnClicked.Tag);
        }
        catch
        {
            return null;
        }
    }
}

// Public method
public void ShowImages(string strDirectory)
{
    Controls.Clear();
tooltip.RemoveAll();

    try
    {
        astrFileNames = Directory.GetFiles(strDirectory);
    }
    catch
    {
        return;
    }

    i = x = y = 0;
timer.Start();
}

// Event handlers
void TimerOnTick(object obj, EventArgs ea)
{
    Image image;

    if (i == astrFileNames.Length)
    {
        timer.Stop();
        return;
    }
    try
    {
        image = Image.FromFile(astrFileNames[i]);
    }
    catch
    {
        i++;
        return;
    }
    int cxImage = image.Width;
    int cyImage = image.Height;

    // Convert image to small size for button.
    SizeF sizef = new SizeF(cxImage / image.HorizontalResolution,
                            cyImage / image.VerticalResolution);
    float fScale = Math.Min(cxButton / sizef.Width,
                             cyButton / sizef.Height);
    sizef.Width *= fScale;
    sizef.Height *= fScale;
    Size size = Size.Ceiling(sizef);
    Bitmap bitmap = new Bitmap(image, size);
    image.Dispose();

    // Create button and add to panel.
    Button btn   = new Button();
    btn.Image    = bitmap;
    btn.Location = new Point(x, y) + (Size) AutoScrollPosition;
    btn.Size     = new Size(cxButton, cyButton);
    btn.Tag      = astrFileNames[i];
    btn.Click    += new EventHandler(ButtonOnClick);
    Controls.Add(btn);

    // Give button a ToolTip.
tooltip.SetToolTip(btn, String.Format("{0}\n{1}x{2}",
Path.GetFileName(astrFileNames[i]),
    cxImage, cyImage));

    // Adjust i, x, and y for next image.

    AdjustXY(ref x, ref y);
    i++;
} void ButtonOnClick(object obj, EventArgs ea)
{
    btnClicked = (Button) obj;

    if (ImageClicked != null)
        ImageClicked(this, EventArgs.Empty);
}
protected override void OnResize(EventArgs ea)
{
    base.OnResize(ea);

    AutoScrollPosition = Point.Empty;
    int x = 0, y = 0;

    foreach (Control cntl in Controls)
    {
        cntl.Location = new Point(x, y) + (Size)
            AutoScrollPosition;
        AdjustXY(ref x, ref y);
    }
}
void AdjustXY(ref int x, ref int y)
{
    y += cyButton;

    if (y + cyButton > Height -
        SystemInformation.HorizontalScrollBarHeight)
    {
        y = 0;
        x += cxButton;
    }
}
The `ImagePanel` constructor sets its `AutoScroll` property to `true`. If there are more buttons than can fit in the allotted space for the panel, the scroll bars need to be displayed so that the user can scroll to the other buttons.

`ImagePanel` implements a public method named `ShowImages` that has a single argument specifying a directory name. `ShowImages` is responsible for obtaining an array of all the files in that directory, loading an `Image` object for each file in the directory that doesn't raise an exception when `Image.FromFile` is called, creating a bitmap that contains the image scaled down to the size of the button, creating the button, and also creating a `ToolTip` that has the name of the image and its pixel dimensions.

Actually, that was the first version of the program. It turned out that this job took much too long for directories containing many large bitmaps. My solution was to spread out the job using a `Timer` object. (It's a simple form of multitasking that doesn't require using multiple threads.) The `Timer` object is created as a field of the `ImagePanel` object and given a tick interval of 1 millisecond during the `ImagePanel` constructor. After the `ShowImages` method gets the array of filenames, it initializes a few variables (an index of the filename array and x and y coordinates for the buttons) and starts the timer.

The `Tick` event handler is responsible for calling `Image.FromFile` and creating a button based on that image. Notice that `Image.FromFile` is called for every file in the directory! If `Image.FromFile` returns properly, it successfully loaded an image. If it throws an exception, either the file wasn't an image file supported by the method or the file was corrupted in some way.

You should also notice that this job can be interrupted before the entire directory has been read. Whenever `ShowImages` is called, it clears all the buttons and ToolTips and starts over again with the new directory.

The `ImagePanel` class also implements a public event named `ImageClicked`. This event is triggered whenever one of the buttons is clicked. The two read-only properties `ClickedControl` and `ClickedImage` return the button that was clicked and the image displayed on that button.

Here's the program itself. `ImageDirectory` makes use of `ImagePanel` as well as `DirectoryTreeView`, with a `Splitter` control to separate them on the client area. The constructor also creates an object of type `PictureBoxPlus`, which is a class I created in Chapter 11 that enhances `PictureBox` to provide a `NoDistort` property that maintains the correct aspect ratio when an image is stretched to the size of the control. This `PictureBoxPlus` control is used to display the clicked image stretched to the size of the client area. The control has its `Visible` property initially set to `false`.

```
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageDirectory: Form
{
    PictureBoxPlus    picbox;
    DirectoryTreeView dirtree;
    ImagePanel        imgpanel;
    Splitter          split;
    TreeNode          tnSelect;
    Control           cnt1Clicked;
    Point             ptPanelAutoScroll;
```
public static void Main()
{
    Application.Run(new ImageDirectory());
}

public ImageDirectory()
{
    Text = "Image Directory";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    // Create (invisible) control for displaying large image.
    picbox = new PictureBoxPlus();
    picbox.Parent = this;
    picbox.Visible = false;
    picbox.Dock = DockStyle.Fill;
    picbox.SizeMode = PictureBoxSizeMode.StretchImage;
    picbox.NoDistort = true;
    picbox.MouseDown += new MouseEventHandler(PictureBoxOnMouseDown);

    // Create controls for displaying thumbnails.
    imgpanel = new ImagePanel();
    imgpanel.Parent = this;
    imgpanel.Dock = DockStyle.Fill;
    imgpanel.ImageClicked += new EventHandler(ImagePanelOnImageClicked);

    split = new Splitter();
    split.Parent = this;
    split.Dock = DockStyle.Left;
    split.BackColor = SystemColors.Control;

    dirtree = new DirectoryTreeView();
    dirtree.Parent = this;
    dirtree.Dock = DockStyle.Left;
    dirtree.AfterSelect +=
        new TreeViewEventHandler(DirectoryTreeViewOnAfterSelect);

    // Create menu with one item (Refresh).
    Menu = new MainMenu();
    Menu.MenuItems.Add("&View");
MenuItem mi = new MenuItem("&Refresh",
    new EventHandler(MenuOnRefresh),
    Shortcut.F5);
Menu.MenuItems[0].MenuItems.Add(mi);
}
void DirectoryTreeViewOnAfterSelect(object obj, TreeViewEventArgs tvea)
{
    tnSelect = tvea.Node;
    imgpanel.ShowImages(tnSelect.FullPath);
}
void MenuOnRefresh(object obj, EventArgs ea)
{
    dirtree.RefreshTree();
}
void ImagePanelOnImageClicked(object obj, EventArgs ea)
{
    // Get clicked control and image.
    cntlClicked = imgpanel.ClickedControl;
    picbox.Image = imgpanel.ClickedImage;

    // Save auto-scroll position.
    ptPanelAutoScroll = imgpanel.AutoScrollPosition;
    ptPanelAutoScroll.X *= -1;
    ptPanelAutoScroll.Y *= -1;

    // Hide and disable the normal controls.
    imgpanel.Visible = false;
    imgpanel.Enabled = false;
    imgpanel.AutoScrollPosition = Point.Empty;

    split.Visible = false;
    split.Enabled = false;

    dirtree.Visible = false;
    dirtree.Enabled = false;

    // Make the picture box visible.
    picbox.Visible = true;
}
// Event handlers and method involved with restoring controls
void PictureBoxOnMouseDown(object obj, MouseEventArgs mea) {
    RestoreControls();
}

protected override void OnKeyDown(KeyEventArgs kea) {
    if (kea.KeyCode == Keys.Escape)
        RestoreControls();
}

void RestoreControls() {
    picbox.Visible = false;
    dirtree.Visible = true;
    dirtree.Enabled = true;
    split.Enabled = true;
    split.Visible = true;
    imgpanel.AutoScrollPosition = ptPanelAutoScroll;
    imgpanel.Visible = true;
    imgpanel.Enabled = true;
    cntlClicked.Focus();
}

Whenever the selection in the DirectoryTreeView control changes (indicated by a call to the DirectoryTreeViewOnAfterSelect event handler), the program calls the ShowImages method of the ImagePanel. Here's the program displaying metafiles from one of the directories of Visual Studio .NET.

Whenever one of the buttons is clicked, the program is notified by a call to its ImagePanelOnImageClicked event handler. The event handler responds by making the three visible controls invisible and the invisible control (PictureBoxPlus) visible. The image is stretched to the size of the client area with its aspect ratio maintained:
The client area can be returned to normal by clicking the client area or pressing the Esc key.

**List View Basics**

In its most sophisticated form, the `ListView` control displays textual information in rows and columns with column headings. The first column of information contains the list view items, and the other columns contain subitems associated with each item. For example, in Windows Explorer, the filename is the item, and the file size, modified date, and attributes are all subitems. The `ListView` control can also display the simple list of items themselves (without subitems), the items in multiple columns with small icons, and items in multiple columns with large icons.

A number of different classes are involved in creating a `ListView` object, but let's begin our tour with the `ListView` class itself. `ListView` has several essential properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>View</td>
<td>get/set</td>
</tr>
<tr>
<td>ImageList</td>
<td>SmallImageList</td>
<td>get/set</td>
</tr>
<tr>
<td>ImageList</td>
<td>LargeImageList</td>
<td>get/set</td>
</tr>
<tr>
<td>ListView.ColumnHeaderCollection</td>
<td>Columns</td>
<td>get</td>
</tr>
<tr>
<td>ListView ListViewItemCollection</td>
<td>Items</td>
<td>get</td>
</tr>
</tbody>
</table>

The `View` enumeration contains members for the four different formats in which a `ListView` control can display its data. You're probably familiar with the four options from what you've seen in various menus, including a toolbar button in the standard `OpenFileDialog` and `SaveFileDialog` dialog boxes:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LargeIcon</td>
<td>0</td>
</tr>
<tr>
<td>Details</td>
<td>1</td>
</tr>
<tr>
<td>SmallIcon</td>
<td>2</td>
</tr>
<tr>
<td>List</td>
<td>3</td>
</tr>
</tbody>
</table>

For the `LargeIcon` option, each item is displayed with a large bitmap (generally 48 pixels square) that is one of the images stored in the `LargeImageList` property. For the other `View` options, the item is displayed with a small bitmap (generally 16 pixels square) from the `SmallImageList` property. The images in these two `ImageList` objects must coincide; for example, the third image in `LargeImageList` should be a larger version of the third image in `SmallImageList`.

The `Columns` property is an object of type `ListView.ColumnHeaderCollection`, which is yet another implementation of the `ICollection`, `IEnumerable`, and `IList` interfaces. (There are more to come in this chapter.) Here's a complete list of its properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView.ColumnHeaderCollection</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As you can see, an object of type `ListView.ColumnHeaderCollection` is basically a collection of read-only `ColumnHeader` objects. The class implements the customary `Clear`, `Insert`, `Remove`, `Add`, and `AddRange` methods. Here are those last two methods as implemented in this class:

### `ListView.ColumnHeaderCollection Methods (selection)`

- `int Add(ColumnHeader colhead)`
- `ColumnHeader Add(string strText, int iWidth, HorizontalAlignment ha)`
- `void AddRange(ColumnHeader[] acolheads)`

As you can deduce from the second `Add` implementation, a `ColumnHeader` object is basically some text, an initial width of the column in pixels, and an alignment. You've seen the `HorizontalAlignment` enumeration before:

#### `HorizontalAlignment Enumeration`

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>0</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
</tr>
<tr>
<td>Center</td>
<td>2</td>
</tr>
</tbody>
</table>

The alignment is considered an important element of the column header because it affects not only the text in the column header but also the items or subitems listed in that column as well.

`ColumnHeader` itself has a default constructor and only three read/write properties, which are the same as the arguments to the `Add` method just shown:

### `ColumnHeader Properties (selection)`

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Width</td>
<td>get/set</td>
</tr>
<tr>
<td>HorizontalAlignment</td>
<td>TextAlign</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The only other properties of `ColumnHeader` are read-only, and they indicate the `ListView` object to which the `ColumnHeader` object belongs and the index of that column header among the collection of column headers.

Let's go back to the table of essential `ListView` properties. The last item in the table was a property named `Items`, which is an object of type `ListView.ListViewItemCollection`. Here are its properties:

### `ListView.ListViewItemCollection Properties`

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListViewItem</td>
<td>[]</td>
<td>get/set</td>
</tr>
<tr>
<td>int</td>
<td>Count</td>
<td>get</td>
</tr>
</tbody>
</table>
The `Items` property of `ListView` is basically a collection of `ListViewItem` objects. As usual, we can get a hint of what a `ListViewItem` is by looking at the `Add` and `AddRange` methods of `ListView.ListViewItemCollection`:

### ListView.ListViewItemCollection Methods (selection)

- `ListViewItem Add(ListViewItem lvitem)`
- `ListViewItem Add(string strItem)`
- `ListViewItem Add(string strItem, int indexImage)`
- `void AddRange(ListViewItem[] alvitems)`

The `strItem` argument is the text string associated with the item. Regardless of what view is selected, this text string is always displayed. The `View.Details` option also displays subitems, which we haven't encountered just yet. The `indexImage` argument is an index into both the `LargeImageList` and `SmallImageList` properties of the `ListView` control.

`ListViewItem` has seven different constructors:

### ListViewItem Constructors

- `ListViewItem()`
- `ListViewItem(string strItem)`
- `ListViewItem(string strItem, int indexImage)`
- `ListViewItem(string[] astrItems)`
- `ListViewItem(string[] astrItems, int indexImage)`
- `ListViewItem(string[] astrItems, int indexImage, Color clrFore, Color clrBack, Font font)`
- `ListViewItem(ListViewItem.ListViewSubItem[] aSubItems, int indexImage)`

When you specify an array of strings in the constructor, you're actually specifying an item and one or more subitems associated with that item.

The following properties of the `ListViewItem` class are essential:

### ListViewItem Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td><code>Text</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>ImageIndex</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>object</code></td>
<td><code>Tag</code></td>
<td>get/set</td>
</tr>
<tr>
<td><code>ListViewItem.ListViewSubItemCollection</code></td>
<td><code>SubItems</code></td>
<td>get</td>
</tr>
</tbody>
</table>

The `ListViewItem` object contains text and an image index as well as a `Tag` property that lets you attach arbitrary data to the item. `ListViewItem` also contains a collection of subitems, which are objects of `ListViewItem.ListViewSubItemCollection`. Here's a complete list of properties of that class:
As usual, we can get an insight into what constitutes a subitem by looking at the arguments of the Add and AddRange methods of the class:

**ListViewItem.ListViewSubItemCollection Methods (selection)**

ListViewSubItem Add(string strText)
ListViewSubItem Add(string strText, Color clrFore, Color clrBack, Font font)
ListViewSubItem Add(ListViewItem.ListViewSubItem lvsi)
void AddRange(string[] astrText)
void AddRange(string[] astrText, Color clrFore, Color clrBack, Font font)
void AddRange(ListViewItem.ListViewSubItem[] alvsi)

The **ListViewItem.ListViewSubItem** constructors have similar arguments:

**ListViewItem.ListViewSubItem Constructors**

ListViewItem.ListViewSubItem()
ListViewItem.ListViewSubItem(ListViewItem lviOwner, string strText)
ListViewItem.ListViewSubItem(ListViewItem lviOwner, string strText,
                              Color clrFore, Color clrText, Font font)

The class has only four properties:

**ListViewItem.ListViewSubItem Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>get/set</td>
</tr>
<tr>
<td>Font</td>
<td>Font</td>
<td>get/set</td>
</tr>
<tr>
<td>Color</td>
<td>BackColor</td>
<td>get/set</td>
</tr>
<tr>
<td>Color</td>
<td>ForeColor</td>
<td>get/set</td>
</tr>
</tbody>
</table>

It's now time to put all this information into service. I trust you'll recall the series of SysInfo (system information) programs that were the highlight of Chapter 4 and also showed up in some subsequent chapters. I'd now like to show a version that uses a ListView control. This program also makes use of the *SysInfoReflectionStrings* class that provides several public static properties and methods, the Labels and Values properties being the most important. Both properties return arrays of strings that indicate (respectively) the names of the static properties in the SystemInformation class and their current values. The Count property returns the number of strings in the arrays. The MaxLabelWidth and MaxValueWidth methods return the maximum width of the string in each array. I use those methods in this program to set the initial column widths.

*SysInfoListView.cs*

// ----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoListView: Form
{
    public static void Main()
    {
        Application.Run(new SysInfoListView());
    }
    public SysInfoListView()
    {
        Text = "System Information (List View)";

        // Create ListView control.
        ListView listview = new ListView();
        listview.Parent = this;
        listview.Dock = DockStyle.Fill;
        listview.View = View.Details;

        // Define columns based on maximum string widths.
        Graphics grfx = CreateGraphics();
        listview.Columns.Add("Property",
            (int) SysInfoReflectionStrings.MaxLabelWidth(grfx, Font),
            HorizontalAlignment.Left);
        listview.Columns.Add("Value",
            (int) SysInfoReflectionStrings.MaxValueWidth(grfx, Font),
            HorizontalAlignment.Left);
        grfx.Dispose();

        // Get the data that will be displayed.
        int iNumItems = SysInfoReflectionStrings.Count;
        string[] astrLabels = SysInfoReflectionStrings.Labels;
        string[] astrValues = SysInfoReflectionStrings.Values;

        // Define the items and subitems.
As you can see, despite the tongue-twisting long class names involved with the various ListView collections, the actual code is rather terse. The constructor begins by creating the ListView object, assigning the form as its parent, giving the object a Dock property of DockStyle.Fill, and then setting the View property to View.Details. (Anything else would be meaningless for this program.) This particular ListView object doesn't use any image lists.

Next, the constructor defines the two column headers by calling the three-argument Add method of the Columns property. The items and subitems are added in the for loop at the bottom. In that for loop, the program creates a ListViewItem object based on an element of the astrLabels array. It then uses the Add method of the SubItems property to add a single subitem, which is an element of the astrValues array. The ListViewItem object is then added to the ListView object using the Add method of the Items property.

And here's the result:

![System Information](image)

The scroll bars are provided by default. You'll probably want to experiment with this program a bit to examine the other features that the default ListView provides and also to try out some properties I haven't talked about here.

**List View Events**

When experimenting with the SysInfoListView program, you'll find that you can select an item in the first column using the mouse or the up and down arrow keys. With the Shift key pressed, you can extend the selection to multiple items. You can also select and deselect individual items (without affecting other selected items) by clicking with the mouse while holding down the Ctrl key. (To turn off the default multiselection feature in a ListView object, set the MultiSelect property to false.)

A program may or may not be interested that the user is changing the selection. However, most programs that use a ListView object for something other than simple display purposes will almost definitely be interested in something called item activation. Windows Explorer, for example, launches applications when the user activates an item. By default, activation occurs when the user double-clicks an item or a group of items, or presses the Enter key when one or more items have been selected. However, you can change that behavior by using the following property:

**ListView Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemActivation</td>
<td>Activation</td>
<td>get/set</td>
</tr>
</tbody>
</table>
The `ItemActivation` enumeration has the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>0</td>
</tr>
<tr>
<td>OneClick</td>
<td>1</td>
</tr>
<tr>
<td>TwoClick</td>
<td>2</td>
</tr>
</tbody>
</table>

`Standard` is default. Both the `OneClick` and `TwoClick` options cause items to change color as the mouse cursor moves over them. The `OneClick` option requires one click for activation; the `TwoClick` option requires two clicks.

Here are the three most important events implemented by the `ListView` class:

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectedIndexChanged</td>
<td>OnSelectedIndexChanged</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>ItemActivate</td>
<td>OnItemActivate</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
<tr>
<td>ColumnClick</td>
<td>OnColumnClick</td>
<td>ColumnClick-EventHandler</td>
<td>ColumnClickEventArgs</td>
</tr>
</tbody>
</table>

None of the other classes associated with the `ListView` control implement any events. The other `ListView` events involve editing, checking, and dragging items.

`ListView` also supports all the events implemented in `Control`. If, for example, a program wants to customize and display a context menu depending on what item the user is clicking with the right mouse button, it can install a `MouseDown` event handler and determine what item the user is clicking by calling the `GetItemAt` method of `ListView`.

When the user clicks a column heading, the `ColumnClick` event is accompanied by the following information:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Column</td>
<td>get</td>
</tr>
</tbody>
</table>

The `SelectedIndexChanged` and `ItemActivate` events aren't accompanied by any information. The program handling these events will want to use the following two properties of `ListView` to obtain the currently selected items:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView.SelectedItemsCollection</td>
<td>SelectedIndices</td>
<td>get</td>
</tr>
<tr>
<td>ListView.SelectedItemsCollection</td>
<td>SelectedItems</td>
<td>get</td>
</tr>
</tbody>
</table>

Yes, these are yet two more collections! The first is just a read-only collection of integers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>[]</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Count</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>IsReadOnly</td>
<td>get</td>
</tr>
</tbody>
</table>
The `Add` and `AddRange` methods of this class are not public. The second collection has the following properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ListView</code></td>
<td><code>SelectedItemCollection</code></td>
<td></td>
</tr>
</tbody>
</table>

Again, the `Add` and `AddRange` methods are not public. To initialize items programatically, use the following property of `ListViewItem`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td><code>Selected</code></td>
<td><code>get/set</code></td>
</tr>
</tbody>
</table>

You can also use this property instead of `SelectedIndices` or `SelectedItems` to obtain the selected items. You'll need to loop through all the items of the `ListView` object and check which ones have the `Selected` property set.

The following class, `FileListView`, derives from `ListView` to display a list of files stored in a given directory. Unlike the list view in Windows Explorer, `FileListView` doesn't display subdirectories along with files.

```csharp
using System;
using System.Diagnostics; // For Process.Start
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class FileListView: ListView
{
    string strDirectory;

    public FileListView()
    {
        View = View.Details;

        // Get images for file icons.
        ImageList imglst = new ImageList();
        imglst.Images.Add(new Bitmap(GetType(), "DOC.BMP"));
        imglst.Images.Add(new Bitmap(GetType(), "EXE.BMP"));

        SmallImageList = imglst;
        LargeImageList = imglst;
```
// Create columns.
Columns.Add("Name", 100, HorizontalAlignment.Left);
Columns.Add("Size", 100, HorizontalAlignment.Right);
Columns.Add("Modified", 100, HorizontalAlignment.Left);
Columns.Add("Attribute", 100, HorizontalAlignment.Left);
}

public void ShowFiles(string strDirectory)
{
    // Save directory name as field.
    this.strDirectory = strDirectory;
    Items.Clear();
    DirectoryInfo dirinfo = new DirectoryInfo(strDirectory);
    FileInfo[] afileinfo;

    try
    {
        afileinfo = dirinfo.GetFiles();
    }
    catch
    {
        return;
    }

    foreach (FileInfo fi in afileinfo)
    {
        // Create ListViewItem.
        ListViewItem lvi = new ListViewItem(fi.Name);

        // Assign ImageIndex based on filename extension.
        if (Path.GetExtension(fi.Name).ToUpper() == ".EXE")
            lvi.ImageIndex = 1;
        else
            lvi.ImageIndex = 0;

        // Add file length and modified time subitems.
        lvi.SubItems.Add(fi.Length.ToString("N0"));
        lvi.SubItems.Add(fi.LastWriteTime.ToString());
// Add attribute subitem.

string strAttr = "";

if ((fi.Attributes & FileAttributes.Archive) != 0)
    strAttr += "A";

if ((fi.Attributes & FileAttributes.Hidden) != 0)
    strAttr += "H";

if ((fi.Attributes & FileAttributes.ReadOnly) != 0)
    strAttr += "R";

if ((fi.Attributes & FileAttributes.System) != 0)
    strAttr += "S";

lvi.SubItems.Add(strAttr);

    // Add completed ListViewItem to FileListView.

Items.Add(lvi);
}
}

protected override void OnItemActivate(EventArgs ea)
{
    base.OnItemActivate(ea);

    foreach (ListViewItem lvi in SelectedItems)
    {
        try
        {
        }
        catch
        {
            continue;
        }
    }
}
Windows Explorer probably uses the API function `ExtractAssociatedIcon` to obtain an image for each file it displays. However, that facility isn’t exposed in the Windows Forms classes. To provide some sample images anyway, `FileListView` loads two bitmaps that I copied from the Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Bitmaps\Outline\NoMask directory. Both the `SmallImageList` and `LargeImageList` properties get the same pair of small images. The constructor concludes by creating four columns.

The `FileListView` class implements a public `ShowFiles` method that does most of the work of the class. The method creates an object of type `DirectoryInfo` based on the specified directory and then gets an array of `FileInfo` structures by calling the `GetFiles` method. Each member of the array becomes an item and three subitems. Much of the code is devoted to formatting the items. If the filename extension is `.exe`, the `ImageIndex` property is set to 1 for the Exe.bmp image; otherwise, it’s set to 0 for the Doc.bmp image. (I know: What about `.dll` files? What about `.com` files? You’re welcome to enhance the images if you wish.)

`FileListView` also overrides the `OnItemActivate` method. For each selected item, the program calls the static `Process.Start` method. If the file is an executable, the file will be launched directly. If the file is a document with a known association, the associated file will be launched with the document.

To see what this custom `ListView` control looks like, we need a Windows Explorer–like program that combines both `DirectoryTreeView` and `FileListView`. `ExplorerLike` is such a program.

```csharp
using System;
using System.Drawing;
using System.Windows.Forms;

class ExplorerLike: Form
{
    FileListView      filelist;
    DirectoryTreeView dirtree;
    MenuItemView      mivChecked;

    public static void Main()
    {
        Application.Run(new ExplorerLike());
    }
    public ExplorerLike()
    {
        Text = "Windows Explorer-Like Program";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
}
```
// Create controls.

filelist = new FileListView();
filelist.Parent = this;
filelist.Dock = DockStyle.Fill;

Splitter split = new Splitter();
split.Parent = this;
split.Dock = DockStyle.Left;
split.BackColor = SystemColors.Control;

dirtree = new DirectoryTreeView();
dirtree.Parent = this;
dirtree.Dock = DockStyle.Left;
dirtree.AfterSelect +=
    new
    TreeViewEventHandler(DirectoryTreeViewOnAfterSelect);

// Create View menu.

Menu = new MainMenu();
Menu.MenuItems.Add("&View");

string[] astrView = { "La""rg""e Icons", "S&mall Icons", "&List", "&Details" };
View[] aview = { View.LargeIcon, View.SmallIcon, View.List, View.Details };
EventHandler eh = new EventHandler(MenuOnView);
for (int i = 0; i < 4; i++)
{
    MenuItemView miv = new MenuItemView();
    miv.Text = astrView[i];
    miv.View = aview[i];
    miv.RadioCheck = true;
    miv.Click += eh;

    if (i == 3) // Default == View.Details
    {
        mivChecked = miv;
        mivChecked.Checked = true;
        filelist.View = mivChecked.View;
    }
}
Menu.MenuItems[0].MenuItems.Add(miv);
}
Most of this program is devoted to processing menu commands that let you change the View property of the FileListView control and refresh the DirectoryTreeView contents. The only connection between the two controls is implemented in the DirectoryTreeViewOnAfterSelect event handler, which calls the ShowFiles method of FileListView with the newly selected directory. Here's the program showing part of the Windows system directory:
Chapter 23: Metafiles

Overview

Metafiles are to vector graphics as bitmaps are to raster graphics. While bitmaps generally originate from real-world images, metafiles are usually constructed by humans in collaboration with computer programs. A metafile consists of a series of binary records that correspond to graphics function calls—to draw lines, curves, filled areas, and text. Metafiles can also contain embedded bitmaps. A metafile can be stored in a disk file or can reside entirely in memory.

Paint programs create bitmaps; drawing programs create metafiles. In a well-designed drawing program, you can easily grab on-screen graphical objects with the mouse and move them somewhere else. That's because all the individual components of the picture are stored as separate records. In a paint program, such feats aren't possible without a lot of heavy image analysis. Paint programs generally restrict you to moving or inserting rectangular chunks of the bitmap.

Because a metafile describes an image in terms of graphical drawing commands, the metafile image can be scaled in size without loss of resolution. Bitmaps don't work that way; if you display a bitmap at twice the size, you don't get twice the resolution. The bits in the bitmap are simply replicated horizontally or vertically. Any smoothing that might be imposed on the display might eliminate jaggies, but at the cost of making the image fuzzier.

A metafile can be converted to a bitmap, but with some loss of information: the graphical objects that make up the metafile are no longer separate and become blended together in one big image. Converting bitmaps to metafiles is a much more difficult job, usually restricted to very simple images and requiring a lot of processing power to analyze edges and outlines. However, as I noted earlier, a metafile can contain an embedded bitmap.

Metafiles are used most often these days for sharing pictures among programs through the clipboard and for clip art. Because metafiles describe a picture as a collection of graphics function calls, they generally take up much less space and are more device independent than bitmaps.

However, rendering a metafile can be slower than rendering a bitmap containing the same image. A bitmap of a particular size and color format takes the same time to display regardless of the complexity of the image. The time it takes to display a metafile is directly related to the number of drawing commands it contains.

Don't confuse metafiles with graphics paths! A path is simply a collection of coordinates; a metafile includes specifications of pens and brushes as well. A path stores text as a series of character outlines; a metafile stores the arguments to the actual DrawString call. There are no standard formats for saving paths to files or passing them through the clipboard. Metafiles have been designed to be saved as files and passed through the clipboard. (I'll discuss using metafiles with the clipboard in Chapter 24.)

Metafiles have been supported under Windows since version 1.0 (1985). The original metafile format is now referred to as the Windows Metafile and is associated with a filename extension of .wmf. Metafiles were enhanced with the introduction of the 32-bit versions of Windows. The 32-bit versions of Windows continued to support the old metafile format and also introduced a new metafile format, called the Enhanced Metafile and associated with a filename extension of .emf.

The GDI+ graphics system in Windows Forms introduces a number of new drawing commands, and these commands affect metafiles as well. Enhanced metafiles that contain GDI+ drawing commands are referred to as EMF+ ("EMF plus") metafiles, but the filename extension is still .emf. It's also possible to create metafiles from a Windows Forms program that are compatible with the original EMF format and readable by regular old 32-bit Windows programs.

Loading and Rendering Existing Metafiles

You already know from Chapter 11 how to load and display metafiles. You can use the same static FromFile method of the Image class to load a metafile from disk just as you can load a bitmap image from disk:

```csharp
Image image = Image.FromFile("PrettyPicture.emf");
```
You can also display this metafile in the same way you display a bitmap. Use one of the many `DrawImage` or `DrawImageUnscaled` methods of the `Graphics` class:

```csharp
grfx.DrawImage(image, x, y);
```

If you have any WMF or EMF files on your hard drive (and it's likely you do if you've installed any application that has a clip art library), you can use the ImageIO program from Chapter 16 to load and display those metafiles. Even Visual Studio .NET comes with a collection of metafiles located by default in the C:\Program Files\Microsoft Visual Studio .NET\Common7\Graphics\Metafile directory.

Metafiles are considered to be images because—like the `Bitmap` class—the `Metafile` class is descended from `Image`:

![Object](image)

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>MarshalByRefObject</td>
</tr>
<tr>
<td>Image (abstract)</td>
</tr>
<tr>
<td>Bitmap</td>
</tr>
<tr>
<td>Metafile</td>
</tr>
</tbody>
</table>

Also like `Bitmap`, the `Metafile` class is sealed and hence can't be inherited. Although the `Image` and `Bitmap` classes are defined in the `System.Drawing` namespace, `Metafile` and its related classes and enumerations are defined in the `System.Drawing.Imaging` namespace. If you call `GetType` on the return value of `ImageFromFile`, you'll get a type of either `System.Drawing.Bitmap` or `System.Drawing.Imaging.Metafile`.

Watch out for the terminology involving metafiles and `Metafile` objects. A metafile is a collection of drawing commands that can exist in a disk file or in memory. A `Metafile` object is an instance of the `Metafile` class. The static `FromFile` method of the `Image` class creates a `Metafile` object based on an existing metafile.

The bulk of the `Metafile` class is its 39 constructors, but some constructors are much simpler than others. To create a `Metafile` object from an existing metafile referenced by either a filename or a `Stream` object, you can use the following two constructors:

**Metafile Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Metafile(string strFileName)</code></td>
</tr>
<tr>
<td><code>Metafile(Stream stream)</code></td>
</tr>
</tbody>
</table>

These two constructors are essentially equivalent to the corresponding static `FromFile` methods of the `Image` class except (of course) the constructors explicitly return an object of type `Metafile`:

```csharp
Metafile mf = new Metafile("PrettyPicture.emf");
```

Because `Metafile` is descended from `Image`, you use the same methods for displaying the metafile:

```csharp
grfx.DrawImage(mf, x, y);
```

Indeed, you can do just about anything with the metafile that is supported by the `Image` class. If you've loaded an existing metafile from a file or stream, however, you can't use the static `FromImage` method of the `Graphics` class to obtain a `Graphics` object for drawing on the metafile. That method is reserved for metafiles that you create anew in your programs.

**Metafile Sizes and Rendering**

As you'll recall from Chapter 11, the `Image` class has several properties that describe the image. Because `Metafile` is descended from `Image`, these properties also apply to metafiles. In particular, you'll find the following properties useful when working with metafiles:
For **Bitmap** objects, the *Size*, *Width*, and *Height* properties indicate the pixel dimension of the bitmap—the number of rows and columns of bits. The *HorizontalResolution* and *VerticalResolution* properties report information that’s probably encoded in the bitmap: the number of pixels per inch horizontally and vertically. You can easily calculate a metrical size of the bitmap in inches by dividing the *Width* by the *HorizontalResolution* and the *Height* by the *VerticalResolution*. If you convert those dimensions to millimeters and multiply by 100, you'll get numbers equal to the *PhysicalDimension* property, which is the size of the bitmap in units of hundredths of millimeters.

For **Metafile** objects, the *Size*, *Width*, and *Height* properties are a little different. In many cases, these properties reflect the extents of the coordinates and sizes of all the objects in the metafile. For example, if the metafile consisted of a single *DrawLine* call with endpoint coordinates of (−50, 25) and (100, 250), the *Width* would probably be 150 (or thereabouts) and the *Height* would be 225 (or so). However, as we’ll see shortly, the creator of the metafile can set the *Width* and *Height* properties to something different. Also, wide lines could affect the size of the image and hence the *Width* and *Height* properties. So, even though metafiles don’t have pixels, they have something equivalent to a pixel size.

*Metafile* objects also have valid *HorizontalResolution* and *VerticalResolution* properties that indicate how the coordinates of the metafile relate to inches. That hypothetical metafile with a single *DrawLine* call might have *HorizontalResolution* and *VerticalResolution* values of 75, so the image would be 2 inches wide and 3 inches high. The *PhysicalDimension* property would be (5080, 7620).

To display a metafile in its metrical size with the upper left corner at the point *(x, y)*, use

```
DrawImage(mf, x, y);
```

or one of the *DrawImage* variants that uses *Point* or *PointF* arguments. The displayed size of the image is not affected by the page transform but is affected by the world transform.

The following *DrawImage* method—and its variants using *Rectangle* and *RectangleF* arguments—displays a metafile stretched to the rectangle:

```
DrawImage(mf, x, y, cx, cy);
```

Both the page transform and the world transform affect the interpretation of the *x, y, cx, and cy* arguments. To display a metafile in its pixel size, set page units to pixels and use

```
DrawImage(mf, x, y, mf.Width, mf.Height);
```

The **Metafile** class has no additional public properties beyond what it inherits from the **Image** class. However, the metafile itself has a header that provides additional information about the metafile. The metafile header is encapsulated in the **MetafileHeader** class. You can obtain an object of **MetafileHeader** using the following instance method:

```
MetafileHeader GetMetafileHeader()
```
Or, for a metafile for which you don't have a Metafile object, you can use one of the following static methods:

**Metafile Static GetMetafileHeader Methods (selection)**

MetafileHeader GetMetafileHeader(string strFileName)
MetafileHeader GetMetafileHeader(Stream stream)

There are two additional static GetMetafileHeader methods for use with Win32 handles to a metafile or an enhanced metafile.

The MetafileHeader class has 10 read-only properties. Here are 5 of them:

**MetafileHeader Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetafileType</td>
<td>Type</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Version</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>MetafileSize</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>EmfPlusHeaderSize</td>
<td>get</td>
</tr>
<tr>
<td>MetaHeader</td>
<td>WmfHeader</td>
<td>get</td>
</tr>
</tbody>
</table>

The Type property indicates the type of the metafile based on the MetafileType enumeration:

**MetafileType Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>0</td>
</tr>
<tr>
<td>Wmf</td>
<td>1</td>
</tr>
<tr>
<td>WmfPlaceable</td>
<td>2</td>
</tr>
<tr>
<td>Emf</td>
<td>3</td>
</tr>
<tr>
<td>EmfPlusOnly</td>
<td>4</td>
</tr>
<tr>
<td>EmfPlusDual</td>
<td>5</td>
</tr>
</tbody>
</table>

The Wmf members identify the old 16-bit metafiles. An Emf metafile was created by a 32-bit Windows program using the Windows API or MFC. By default (as we'll see), the metafiles created by a Windows Forms program are of type EmfPlusDual, which means that they contain both GDI and GDI+ records. Such metafiles are usable by Win32 programs. An EmfPlusOnly metafile contains only GDI+ records and is usable only by Windows Forms programs.

The MetafileSize property indicates the actual storage size of the entire metafile. For metafiles stored on disk, it's equal to the file size. For WMF types, the WmfHeader property has additional information about the metafile.

The following are all the methods of MetafileHeader, which mostly provide a Boolean interface to the Type property:

**MetafileHeader Methods**

```csharp
bool IsWmf()
bool IsWmfPlaceable()
bool IsEmf()
bool IsEmfPlus()
```
bool IsEmfPlusOnly()
bool IsEmfPlusDual()
bool IsEmfOrEmfPlus()
bool IsDisplay()


As we'll see, a metafile is always created based on a particular graphics output device. The IsDisplay method returns true for a metafile based on the video display and false for a metafile based on a printer.

These are the remaining MetafileHeader properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>Bounds</td>
<td>get</td>
</tr>
<tr>
<td>float</td>
<td>DpiX</td>
<td>get</td>
</tr>
<tr>
<td>float</td>
<td>DpiY</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>LogicalDpiX</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>LogicalDpiY</td>
<td>get</td>
</tr>
</tbody>
</table>

The Width and Height of the Bounds property should agree with the Width and Height properties that the Metafile object inherits from Image. The DpiX and DpiY properties should agree with the HorizontalResolution and VerticalResolution properties of the Image class. The LogicalDpiX and LogicalDpiY properties don't have any relevance for Windows Forms programs, and you should ignore them.

The X and Y properties of the Bounds property aren't necessarily 0. For example, earlier I discussed a hypothetical metafile that consisted of a sole DrawLine call with coordinates of (−50, 25) and (100, 250). The Bounds property of the metafile header is generally the smallest rectangle that encloses all the graphical objects in the metafile. A simple calculation predicts that the Bounds property will be the rectangle (−50, 25, 150, 225).

Actually, in this case, you're more likely to see a Bounds property of (−51, 24, 153, 228). Because GDI+ draws lines up to and including the second point, the line is actually a pixel longer than simple arithmetic would dictate. Also, the line has a finite width, which increases the total dimension by another pixel on either end. Moreover, the program creating the metafile can set a Bounds property other than what the contents of the metafile would imply.

The origin of the Bounds rectangle—that is, its X and Y properties—doesn't affect the positioning of the metafile when you render it. For example, if you draw the hypothetical simple metafile I've been discussing using the call

gfx.DrawImage(mf, 0, 0);

you'll see the whole metafile. The upper left corner of the Bounds rectangle is displayed at the point specified in the DrawImage call, in this example, the point (0, 0).

Here's a program that has an OpenFileDialog object configured to load disk-based metafiles and display them.

MetafileViewer.cs

// MetafileViewer.cs © 2001 by Charles Petzold
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Printing;
using System.IO;              // For Path class
using System.Windows.Forms;

class MetafileViewer: Form
{
    protected Metafile mf;
    protected string strProgName;
    protected string strFileName;
    MenuItem miFileSaveAs, miFilePrint,
               miFileProps, miViewChecked;

    public static void Main()
    {
        Application.Run(new MetafileViewer());
    }

    public MetafileViewer()
    {
        Text = strProgName = "Metafile Viewer";
        ResizeRedraw = true;

        Menu = new MainMenu();

        // File menu

        MenuItem mi = new MenuItem("&File");
        mi.Popup += new EventHandler(MenuFileOnPopup);
        Menu.MenuItems.Add(mi);

        // File Open menu item

        mi = new MenuItem("&Open...");
        mi.Click += new EventHandler(MenuFileOpenOnClick);
        mi.Shortcut = Shortcut.CtrlO;
        Menu.MenuItems[0].MenuItems.Add(mi);

        // File Save As Bitmap menu item

        miFileSaveAs = new MenuItem("Save &As Bitmap...");
        miFileSaveAs.Click += new EventHandler(MenuFileSaveAsOnClick);
        Menu.MenuItems[0].MenuItems.Add(miFileSaveAs);
Menu.MenuItems[0].MenuItems.Add("-");

    // File Print menu item

miFilePrint = new MenuItem("&Print...");
miFilePrint.Click += new EventHandler(MenuFilePrintOnClick);
Menu.MenuItems[0].MenuItems.Add(miFilePrint);
Menu.MenuItems[0].MenuItems.Add("-");

    // File Properties menu item

miFileProps = new MenuItem("Properties...");
miFileProps.Click += new EventHandler(MenuFilePropsOnClick);
Menu.MenuItems[0].MenuItems.Add(miFileProps);

    // Edit menu (temporary until Chapter 24)

Menu.MenuItems.Add("&Edit");

    // View menu

Menu.MenuItems.Add("&View");

string[] astr = { "&Stretched to Window",
                 "&Metrical Size", "&Pixel Size" };
EventHandler eh = new EventHandler(MenuViewOnClick);
foreach (string str in astr)
    Menu.MenuItems[2].MenuItems.Add(str, eh);

miViewChecked = Menu.MenuItems[2].MenuItems[0];
miViewChecked.Checked = true;
}
void MenuFileOnPopup(object obj, EventArgs ea)
{
    miFileSaveAs.Enabled =
    miFilePrint.Enabled =
    miFileProps.Enabled = (mf != null);
}
void MenuFileOpenOnClick(object obj, EventArgs ea)
{
    OpenFileDialog dlg = new OpenFileDialog();

dlg.Filter = "All Metafiles|*.wmf;*.emf|"
             +
             "Windows Metafile (*.wmf)|*.wmf|" +
if (dlg.ShowDialog() == DialogResult.OK)
{
    try
    {
        mf = new Metafile(dlg.FileName);
    }
    catch (Exception exc)
    {
        MessageBox.Show(exc.Message, strProgName);
        return;
    }
    strFileName = dlg.FileName;
    Text = strProgName + " - " + Path.GetFileName(strFileName);
    Invalidate();
}
}
protected virtual void MenuFileSaveAsOnClick(object obj, EventArgs ea)
{
    MessageBox.Show("Not yet implemented!", strProgName);
}
void MenuFilePrintOnClick(object obj, EventArgs ea)
{
    PrintDocument prndoc = new PrintDocument();

    prndoc.DocumentName = Text;
    prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
    prndoc.Print();
}
void MenuFilePropsOnClick(object obj, EventArgs ea)
{
    MetafileHeader mh = mf.GetMetafileHeader();

    string str =
    "Image Properties" +
    "\n\tSize = " + mf.Size +
    "\n\tHorizontal Resolution = " + mf.HorizontalResolution +
    "\n\tVertical Resolution = " + mf.VerticalResolution +
    "\n\tPhysical Dimension = " + mf.PhysicalDimension +
    "\n\tMetafile Header Properties" +
    "\n\tBounds = " + mh.Bounds +
    "\n\tDpiX = " + mh.DdpiX +
    "\n\tDpiY = " + mh.DdpiY +
"\n\tLogicalDpiX = " + mh.LogicalDpiX +
"\n\tLogicalDpiY = " + mh.LogicalDpiY +
"\n\tType = " + mh.Type +
"\n\tVersion = " + mh.Version +
"\n\tMetafileSize = " + mh.MetafileSize;

MessageBox.Show(str, Text);
}
void MenuViewOnItemClick(object obj, EventArgs ea)
{
    miViewChecked.Checked = false;
    miViewChecked = (MenuItem) obj;
    miViewChecked.Checked = true;
    Invalidate();
}
void OnPrintPage(object obj, PrintPageEventArgs ppea)
{
    Graphics grfx = ppea.Graphics;
    Rectangle rect = new Rectangle(
        ppea.MarginBounds.Left -
        (ppea.PageBounds.Width -
        (int) grfx.VisibleClipBounds.Width) / 2,
        ppea.MarginBounds.Top -
        (ppea.PageBounds.Height -
        (int) grfx.VisibleClipBounds.Height) / 2,
        ppea.MarginBounds.Width,
        ppea.MarginBounds.Height);

    DisplayMetafile(grfx, rect);
}
protected override void OnPaint(PaintEventArgs pea)
{
    if (mf != null)
        DisplayMetafile(pea.Graphics, ClientRectangle);
}
void DisplayMetafile(Graphics grfx, Rectangle rect)
{
    switch (miViewChecked.Index)
    {
    case 0:  grfx.DrawImage(mf, rect);  break;
    case 1:  grfx.DrawImage(mf, rect.X, rect.Y);  break;
                        mf.Height);
        break;
    }
}
This program has a couple features that go beyond the ImageOpen program from Chapter 16 (which also can load and display metafiles). First, it has a Properties item on the File menu that displays the most important information about the metafile from the Image and MetafileHeader properties. It also has a menu item to select three different ways of displaying the metafile using DrawImage. The Print option lets you print the metafile based on that menu selection.

Several menu items are not yet implemented. The program has an entire Edit menu that I'll show code for in Chapter 24. The File menu also has an unimplemented Save As Bitmap item that I'll go over in the next section.

Converting Metafiles to Bitmaps

I mentioned earlier that it's easy to convert a metafile into a bitmap. In fact, the facility is built into Windows Forms. If you use the ImageIO program to load a metafile, you can save it as a bitmap.

It's possible that you'll want to perform this conversion yourself, either to maintain more control over the process or when you don't want to save the bitmap to a disk file. Perhaps you're dealing with a metafile that contains lots of drawing commands and converting it to a bitmap would speed up the display.

The following program inherits from MetafileViewer and includes a method called MetafileToBitmap that converts a Metafile object to a Bitmap object. The program ends up saving the bitmap to a disk file anyway (just as ImageIO does), but you can use the method for other purposes as well.

```
MetafileConvert.cs
//------------------------------
// MetafileConvert.cs © 2001 by Charles Petzold
//------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;              // For Path class
using System.Windows.Forms;

class MetafileConvert: MetafileViewer
{
    public new static void Main()
    {
        Application.Run(new MetafileConvert());
    }
    public MetafileConvert()
    {
        Text = strProgName = "Metafile Convert";
    }
    protected override void MenuFileSaveAsOnClick(object obj, EventArgs ea)
    {
        SaveFileDialog dlg = new SaveFileDialog();

        if (strFileName != null && strFileName.Length > 0)
        {
            dlg.InitialDirectory = Path.GetDirectoryName(strFileName);
        }
        dlg.Filter = "Bitmaps (.bmp)|*.bmp|All Files|*.*";
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            Bitmap b = TransformMetafile(strFileName);
            try
            {
                File.WriteAllBytes(dlg.FileName, b.ToByteArray());
            }
            catch
            {
                MessageBox.Show("Failed to write bitmap to file.", strProgName);
            }
        }
    }
}
```
To convert a metafile to a bitmap, you must first create a Bitmap object of a particular size. Then you obtain a Graphics object to draw on the bitmap:

```csharp
gfx = Graphics.FromImage(bm);
gfx.DrawImage(mf, 0, 0, cx, cy);
gfx.Dispose();
```

If the size of the bitmap is stored in the variables `cx` and `cy`, you can cover the bitmap with the metafile by using the following code:

```csharp
gfx.DrawImage(mf, 0, 0, cx, cy);
```
These two lines of code will work regardless of the size of the bitmap you create. If you need to display a metafile image on a button, for example, you'll want to make the bitmap the size of the button.

The MetafileToBitmap method in the MetafileConvert program creates a bitmap that's based on the metrical size of the metafile. The Bitmap constructor used in this method includes a Graphics object, in this case a Graphics object for the video display. The bitmap will thus have the same resolution as the video display. From that resolution—and the size and resolution of the metafile—it's easy to calculate the pixel size of the bitmap.

The bitmap that MetafileToBitmap creates is the same size as the bitmap the Save method of Image creates when it converts a metafile to a bitmap.

Another approach to converting a metafile to a bitmap is so simple and straightforward that it eluded me until this book was almost ready to go to press. As you'll recall, one of the Bitmap constructors takes a single Image argument. That argument can be a Metafile object:

```csharp
Bitmap bm = new Bitmap(mf);
```

The resultant bitmap has the same pixel size as the original metafile but with the resolution of the video display.

**Creating New Metafiles**

So far, I've discussed only 2 of the 39 Metafile constructors. Those two constructors are the only ones that directly load an existing metafile based on a filename or a Stream. Three of the Metafile constructors create a metafile object based on Win32 metafile handles and are useful for interfacing with existing code. The remaining 34 constructors to Metafile create a new metafile, which very often means that the constructors create a new disk file or delete the contents of an existing file in preparation for creating a new metafile. Here are the two simplest constructors that create a new metafile:

**Metafile Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metafile(string strFileName, IntPtr ipHdc)</td>
</tr>
<tr>
<td>Metafile(Stream stream, IntPtr ipHdc)</td>
</tr>
</tbody>
</table>

That second argument is quite an oddity. The metafile needs to be associated with a particular graphics output device in order to obtain resolution information. It would make more sense for the second argument to be a Graphics object, like so:

```csharp
Metafile mf = new Metafile("NewFile.emf", grfx); // Wrong, unfortunately!
```

This functionality would match that of the Bitmap constructor in the MetafileToBitmap method I just described.

Instead, the second argument to the Metafile constructor is defined as a Win32 device context handle. The Graphics object encapsulates the Win32 device context, so you need to use the GetHdc and ReleaseHdc methods of the Graphics class to obtain and release this handle:

```csharp
IntPtr ipHdc = grfx.GetDC();
Metafile mf = new Metafile("NewFile.emf", ipHdc);
grfx.ReleaseDC(ipHdc);
```

In most cases, you'll want to create a metafile that is based on the resolution of the video display, but you won't be creating a new metafile in your OnPaint method. Instead, as you know, in any class descended from Control, you can use the CreateGraphics method to obtain such a Graphics object. You should call Dispose on this Graphics object after you're finished using it, so your creation of a new metafile will look something like this:

```csharp
Graphics grfxVideo = CreateGraphics();
IntPtr ipHdc = grfxVideo.GetDC();
```
Metafile mf = new Metafile("NewFile.emf", ipHdc);
grfxVideo.ReleaseDC(ipHdc);
grfxVideo.Dispose();

This code is certainly wordier than equivalent Win32 code. One option of the Win32 metafile-creation function lets you specify a *null* device to indicate the video display, but unfortunately, that option isn't allowed in the *Metafile* constructor.

After you've created the metafile, you must obtain another *Graphics* object to insert drawing commands in the metafile. You obtain this *Graphics* object by using the same static method you use for drawing on a bitmap:
Graphics grfxMetafile = Graphics.FromImage(mf);

For purposes of clarity, I've given these two different *Graphics* objects different names. But because they don't overlap, you can use just one *Graphics* variable if that's convenient.

With the *Graphics* object obtained from the *FromImage* method, you can call any drawing method. A coded record of each method ends up in the metafile. The *Graphics* class also has a method that can be used only with metafiles:

*Graphics Methods (selection)*

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void AddMetafileComment(byte[] abyData)</code></td>
</tr>
</tbody>
</table>

After drawing on the metafile, you'll also want to call *Dispose* on this second *Graphics* object:
grfxMetafile.Dispose();

Here's a small program that inherits from *PrintableForm* and creates a metafile in its constructor. In its *DoPage* method, the CreateMetafile program obtains the size of the metafile and uses that to display multiple copies in its client area or on the printer page.

*CreateMetafile.cs*

//-------------------------------
// CreateMetafile.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;              // Not used for anything yet!
using System.Windows.Forms;

class CreateMetafile: PrintableForm
{
    Metafile mf;

    public new static void Main()
    {
        Application.Run(new CreateMetafile());
    }

    public CreateMetafile()
    {
        Text = "Create Metafile";
    
        // CreateMetafile.cs © 2001 by Charles Petzold
        //-------------------------------
        using System;
        using System.Drawing;
        using System.Drawing.Imaging;
        using System.IO;              // Not used for anything yet!
        using System.Windows.Forms;

        class CreateMetafile: PrintableForm
        {
            Metafile mf;

            public new static void Main()
            {
                Application.Run(new CreateMetafile());
            }

            public CreateMetafile()
            {
                Text = "Create Metafile";
            }
// Create the metafile.

Graphics grfx = CreateGraphics();
IntPtr ipHdc = grfx.GetHdc();

mf = new Metafile("CreateMetafile.emf", ipHdc);
grfx.ReleaseHdc(ipHdc);
grfx.Dispose();

// Draw on the metafile.

grfx = Graphics.FromImage(mf);
grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
grfx.FillEllipse(Brushes.Blue, 20, 20, 20, 20);
grfx.FillEllipse(Brushes.Blue, 60, 20, 20, 20);
grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
grfx.Dispose();

protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    for (int y = 0; y < cy; y += mf.Height)
        for (int x = 0; x < cx; x += mf.Width)
            grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
}

The image contained in the metafile consists of calls to DrawEllipse, FillEllipse, and DrawArc. These calls—with the proper pens and brushes—are rendered on the client area by the program's DoPage method. The program uses DrawImage with width and height arguments to draw the image in its pixel dimension:
The only peculiarity you might observe is that these multiple images aren't pressed up against each other. The margin around each image is one of the side effects of the technique that the *Metafile* class uses to convert floating-point coordinates in GDI+ curves to integer coordinates in GDI metafile records. We'll see shortly how you can control the dimensions of this boundary so that the overindulgent margins go away. 

The CreateMetafile program uses this form of the *DrawImage* method to render the metafile in its pixel (rather than metrical) size:

```csharp
grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
```

For the video display, I could just as easily display the metafile in its metrical size:

```csharp
grfx.DrawImage(mf, x, y);
```

Because the metafile has the same resolution as the video display, the results would be the same. However, the two versions of *DrawImage* would probably render different-sized images on the printer. As with bitmaps, it's easier to accurately position metafiles when displaying them in their pixel size.

An alternative is replacing the entire body of *DoPage* with the following statement:

```csharp
grfx.FillRectangle(new TextureBrush(mf), 0, 0, cx, cy);
```

This statement fills the client area with a *TextureBrush* object created from the metafile. And you probably thought texture brushes could only be created from bitmaps!

Every time you run the CreateMetafile program, it re-creates a file named CreateMetafile.emf. But re-creating that file each time is unnecessary. You might try inserting the following code in the *CreateMetafile* constructor right after the assignment of the *Text* property:

```csharp
if (File.Exists("CreateMetafile.emf"))
{
    mf = new Metafile("CreateMetafile.emf");
    return;
}
```

This code loads the metafile if it exists and then exits from the constructor. (The static *Exists* method of the *File* class is defined in the *System.IO* namespace. The CreateMetafile program conveniently includes a *using* statement for this namespace, even though it's not required for anything else in the program.)

The CreateMetafile program retains the *Metafile* object (named *mf*) as a field. Doing so is necessary for the *DoPage* method to use the metafile that the constructor creates. But it isn't strictly necessary
to save the Metafile object. The DoPage method itself can load the metafile. The CreateMetafileReload program is similar to CreateMetafile where the constructor creates the metafile only if it doesn't exist. But the Metafile object isn't saved as a field. Instead, the filename is stored as a file and the DoPage method loads the metafile itself.

CreateMetafileReload.cs

//---------------------------------------------------
// CreateMetafileReload.cs © 2001 by Charles Petzold
//---------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class CreateMetafileReload: PrintableForm
{
    const string strMetafile = "CreateMetafileReload.emf";

    public new static void Main()
    {
        Application.Run(new CreateMetafileReload());
    }

    public CreateMetafileReload()
    {
        Text = "Create Metafile (Reload)";

        if (!File.Exists(strMetafile))
        {
            // Create the metafile.

            Graphics grfx = CreateGraphics();
            IntPtr ipHdc = grfx.GetHdc();

            Metafile mf = new Metafile(strMetafile, ipHdc);

            grfx.ReleaseHdc(ipHdc);
            grfx.Dispose();

            // Draw on the metafile.

            grfx = Graphics.FromImage(mf);

            grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
            grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
            grfx.FillEllipse(Brushes.Blue, 20, 20, 20, 20);
            grfx.FillEllipse(Brushes.Blue, 60, 20, 20, 20);
        }
    }
}
grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
            grfx.Dispose();
        }
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Metafile mf = new Metafile(strMetafile);

        for (int y = 0; y < cy; y += mf.Height)
            for (int x = 0; x < cx; x += mf.Width)
                grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
    }

    I've been demonstrating the use of Metafile constructors with string arguments indicating filenames. You can also specify a Stream argument. For example, you can replace the constructor in the DoPage method with the following code and it will work the same:

    FileStream fs = new FileStream(strMetafile, FileMode.Open);
    Metafile mf = new Metafile(fs);
    fs.Close();

    What won't work is a FileStream constructor that opens the file for writing only or a FileMode argument that destroys the contents of the file.

    Similarly, you can use a FileStream object in the program's constructor to create the Metafile object:

    FileStream fs = new FileStream(strMetafile, FileMode.Create);
    Metafile mf = new Metafile(fs, ipHdc);

    Notice that this FileMode argument indicates that the file should be re-created. After all the Graphics calls have been made to insert commands into the metafile, close the stream:

    fs.Close();

    It's also possible to use a MemoryStream object to create the metafile in memory. Because MemoryStream objects don't have names, a program must retain either the MemoryStream object or the Metafile object as a field.

    Let's go back to the original CreateMetafile program. If you earlier followed my suggestions by inserting a block of code that called File.Exists, remove it. Now replace the Metafile constructor

    mf = new Metafile("CreateMetafile.emf", ipHdc);

    with this one:

    mf = new Metafile(new MemoryStream(), ipHdc);

    MemoryStream is defined in the System.IO namespace. With this variation of the constructor, the metafile is created and accessed in memory. No file is left behind.

    In the following program, the MemoryStream object (but not the Metafile object) is stored as a field.

    CreateMetafileMemory.cs
    //---------------------------------------------------------------
    // CreateMetafileMemory.cs © 2001 by Charles Petzold
    //---------------------------------------------------------------
    using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class CreateMetafileMemory: PrintableForm
{
    readonly MemoryStream ms = new MemoryStream();

    public new static void Main()
    {
        Application.Run(new CreateMetafileMemory());
    }
    public CreateMetafileMemory()
    {
        Text = "Create Metafile (Memory)";

        // Create the metafile.
        Graphics grfx = CreateGraphics();
        IntPtr ipHdc = grfx.GetHdc();

        Metafile mf = new Metafile(ms, ipHdc);

        grfx.ReleaseHdc(ipHdc);
        grfx.Dispose();

        // Draw on the metafile.
        grfx = Graphics.FromImage(mf);

        grfx.FillEllipse(Brushes.Gray, 0, 0, 100, 100);
        grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
        grfx.FillEllipse(Brushes.Blue, 20, 20, 20, 20);
        grfx.FillEllipse(Brushes.Blue, 60, 20, 20, 20);
        grfx.DrawArc(new Pen(Color.Red, 10), 20, 20, 60, 60, 30, 120);
        grfx.Dispose();
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        ms.Position = 0;
        Metafile mf = new Metafile(ms);

        for (int y = 0; y < cy; y += mf.Height)
            for (int x = 0; x < cx; x += mf.Width)
grfx.DrawImage(mf, x, y, mf.Width, mf.Height);
}

Notice that the first statement of the DoPage method sets the position of the MemoryStream object back to 0. Otherwise, an exception is thrown that is very hard to diagnose.

Creating a metafile in memory is useful if you just need to pass a metafile through the clipboard. You'll see how to use the clipboard to copy and paste metafiles in Chapter 24.

The Metafile Boundary Rectangle

When you insert Graphics drawing commands into a metafile, the metafile calculates a boundary rectangle. This is the smallest rectangle that encompasses all the objects stored in the metafile. You can obtain the width and height of this boundary rectangle by using the Size, Width, and Height properties that the Metafile class inherits from Image. Or you can obtain the complete boundary rectangle from the Bounds property of the MetafileHeader object associated with the metafile.

The programs shown so far in this chapter demonstrate that the metafile often calculates a boundary rectangle that is larger than the contents would imply. If you want total control over the boundary, you can use alternative versions of the Metafile constructor. Here are four versions that let you specify a boundary rectangle when creating a metafile stored in a file:

<table>
<thead>
<tr>
<th>Metafile Constructors (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metafile(string strFileName, IntPtr ipHdc, Rectangle rect)</td>
</tr>
<tr>
<td>Metafile(string strFileName, IntPtr ipHdc, RectangleF rectf)</td>
</tr>
<tr>
<td>Metafile(string strFileName, IntPtr ipHdc, Rectangle rect, MetafileFrameUnit mfu)</td>
</tr>
<tr>
<td>Metafile(string strFileName, IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu)</td>
</tr>
</tbody>
</table>

MetafileFrameUnit is an enumeration defined in the System.Drawing.Imaging namespace. The enumeration indicates the units of the boundary rectangle specified in the constructor. MetafileFrameUnit plays no other role in the metafile, and the argument you specify is not retained:

<table>
<thead>
<tr>
<th>MetafileFrameUnit Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>Pixel</td>
</tr>
<tr>
<td>Point</td>
</tr>
<tr>
<td>Inch</td>
</tr>
<tr>
<td>Document</td>
</tr>
<tr>
<td>Millimeter</td>
</tr>
<tr>
<td>GdiCompatible</td>
</tr>
</tbody>
</table>

If you specify no MetafileFrameUnit argument, the default is GdiCompatible. This is probably not what you want!

For simple metafile creation, the easiest MetafileFrameUnit is definitely Pixel. For example, here's the metafile-creation statement in the original version of CreateMetafile:

mf = new Metafile("CreateMetafile.emf", ipHdc);

Try replacing it with this one:

mf = new Metafile("CreateMetafile.emf", ipHdc,
That rectangle is defined in accordance with the coordinates later passed to the various `Graphics` methods. Now the `Width` and `Height` properties of the `Metafile` object obtained during the `DoPage` method are 101 and 101. Thus, the displayed images are 101 pixels apart:

"Why 101?" you ask. Because the largest object in the metafile was created using this call:

```csharp
grfx.DrawEllipse(Pens.Black, 0, 0, 100, 100);
```

As you'll recall, when the pen is 1 pixel wide, the total width and height of such an object will be 101 pixels.

It isn't necessary for the boundary rectangle to have an origin at (0, 0), nor for the boundary rectangle to accurately describe the coordinates of the drawing methods in the metafile. For example, if you change the rectangle argument to

```csharp
mf = new Metafile("CreateMetafile.emf", ipHdc,
                new Rectangle(-25, -25, 75, 75),
                MetafileFrameUnit.Pixel);
```

the images are displayed like so:
The metafile image has a width and height of 75 pixels, but the origin is (−25, −25). Because no negative coordinates were used in the graphics objects inserted in the metafile, the top and left sides of the image are empty. When displaying the metafile, the image is clipped to the Bounds rectangle.

It’s also possible to specify the rectangle in combination with a Stream argument:

**Metafile Constructors (selection)**

- Metafile(Stream stream, IntPtr ipHdc, Rectangle rect)
- Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf)
- Metafile(Stream stream, IntPtr ipHdc, Rectangle rect, MetafileFrameUnit mfu)
- Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu)

For example, you can replace the metafile constructor in the original version of the CreateMetafile program with the following:

```csharp
mf = new Metafile(new MemoryStream(), ipHdc,
                 new Rectangle(0, 0, 101, 101), MetafileFrameUnit.Pixel);
```

It’s also possible to create a new metafile without specifying either a filename or a Stream:

**Metafile Constructors (selection)**

- Metafile(IntPtr ipHdc, Rectangle rect)
- Metafile(IntPtr ipHdc, RectangleF rectf)
- Metafile(IntPtr ipHdc, Rectangle rect, MetafileFrameUnit mfu)
- Metafile(IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu)

When you specify neither a filename nor a Stream, the metafile is created in memory but you don’t have access to the memory buffer as you do with a MemoryStream object. Here’s another alternative metafile-creation statement for CreateMetafile:

```csharp
mf = new Metafile(ipHdc,
                 new Rectangle(0, 0, 101, 101), MetafileFrameUnit.Pixel);
```
This last set of constructors would seem to imply that it should be possible to create a new metafile by specifying only a device context handle, but such a constructor doesn’t exist.

You can use a `MetafileFrameUnit` other than `Pixel`, but doing so probably makes sense only if you’re also drawing on the metafile in units other than pixels. Let’s examine how the metafile and the page transform interact.

**Metafiles and the Page Transform**

A metafile has a width and a height that are available from the `Size`, `Width`, and `Height` properties of the `Image` class and the `Bounds` property of the `MetafileHeader` class. It is convenient to think of the width and height of the metafile in terms of pixels, much like a bitmap. But the metafile’s width and height are really more closely related to the extents of all the coordinates and sizes used in the graphics functions that went into the metafile.

A metafile also has a resolution in dots per inch that is available from the `HorizontalResolution` and `VerticalResolution` properties of the `Image` class and the `DpiX` and `DpiY` properties of the `MetafileHeader` class. These resolutions indicate how the coordinates and sizes encoded in the metafile correspond to inches.

In addition, a metafile has a metrical dimension, which you can calculate from the pixel dimension and the resolution. Or you can use the `PhysicalDimension` property of the `Image` class to obtain the size in hundredths of millimeters.

The resolution of a metafile is set when the metafile is created. The `Metafile` constructor requires a device context handle, and that output device provides a resolution for the metafile. All the coordinates and sizes in the graphics calls encoded within the metafile must be consistent with that resolution. As you add graphics calls to the metafile, coordinates and sizes are adjusted based on any page transform in effect.

Let’s take a look at a program that creates a metafile containing four overlapping rectangles. Each rectangle is 1-inch square and is drawn with a 1-point-wide pen, but each rectangle is created with a different page transform in effect.

**MetafilePageUnits.cs**

```csharp
//------------------------------------------------
// MetafilePageUnits.cs © 2001 by Charles Petzold
//------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Drawing.Printing;     // Not used for anything yet!
using System.Windows.Forms;

class MetafilePageUnits: PrintableForm
{
    Metafile mf;

    public new static void Main()
    {
        Application.Run(new MetafilePageUnits());
    }
    public MetafilePageUnits()
    {
        Text = "Metafile Page Units";
    }

    void MainFormPaint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        foreach (Rectangle r in rects)
            g.DrawRectangle(Pens.Red, r);
    }
}
```
// Create metafile.

Graphics grfx = CreateGraphics();
IntPtr ipHdc = grfx.GetHdc();

mf = new Metafile("MetafilePageUnits.emf", ipHdc);
grfx.ReleaseHdc(ipHdc);
grfx.Dispose();

// Get Graphics object for drawing on metafile.

grfx = Graphics.FromImage(mf);
grfx.Clear(Color.White);

// Draw in units of pixels (1-point pen width).

grfx.PageUnit = GraphicsUnit.Pixel;
Pen pen = new Pen(Color.Black, grfx.DpiX / 72);
grfx.DrawRectangle(pen, 0, 0, grfx.DpiX, grfx.DpiY);

// Draw in units of 1/100 inch (1-point pen width).

grfx.PageUnit = GraphicsUnit.Inch;
grfx.PageScale = 0.01f;
pen = new Pen(Color.Black, 100f / 72);
grfx.DrawRectangle(pen, 25, 25, 100, 100);

// Draw in units of millimeters (1-point pen width).

grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
pen = new Pen(Color.Black, 25.4f / 72);
grfx.DrawRectangle(pen, 12.7f, 12.7f, 25.4f, 25.4f);

// Draw in units of points (1-point pen width).

grfx.PageUnit = GraphicsUnit.Point;
pen = new Pen(Color.Black, 1);
grfx.DrawRectangle(pen, 54, 54, 72, 72);

gfx.Dispose();
Despite the fact that the four rectangles are drawn in units of pixels, inches, millimeters, and points, it's comforting to see how these rectangles are stored and rendered in a consistent manner:

The first rectangle (the one at the top left) is drawn in units of pixels:

```csharp
gfx.PageUnit = GraphicsUnit.Pixel;
Pen pen = new Pen(Color.Black, gfx.DpiX / 72);
gfx.DrawRectangle(pen, 0, 0, gfx.DpiX, gfx.DpiY);
```

The default `PageUnit` property setting for a metafile based on the video display is `GraphicsUnit.Pixel`, so that first statement isn't strictly required. But it's necessary if you try the variation I'm going to describe next.

So far, all the metafiles we've created have been based on the video display resolution. You can also base a metafile on the printer. Simply replace the statement

```csharp
Graphics gfx = CreateGraphics();
```

near the top of the constructor with these statements that obtain a `PrinterSettings` object for the default printer and then a `Graphics` object suitable for creating a metafile:

```csharp
PrinterSettings prnset = new PrinterSettings();
Graphics gfx = prnset.CreateMeasurementGraphics();
```

As the resultant metafile is rendered on the screen and the printer, it appears to be the same as the original one based on the video display. However, a little exploration will reveal that the new metafile resolution is now consistent with your default printer. In addition, all the coordinates and sizes encoded in the metafile reflect this higher resolution.

**The Metafile Type**

A metafile consists of a header and records. We'll examine how to view metafile records toward the end of this chapter, but for now you should know that each record is identified by a member of the `EmfPlusRecordType` enumeration, which is defined in the `System.Drawing.Imaging` namespace. At 253 members, `EmfPlusRecordType` is the largest enumeration in all of the .NET Framework. Here are three related members of the enumeration:

```csharp
EmfPlusRecordType Enumeration (selection)
```
The third item in this little list has a prefix of \textit{Wmf}, which stands for Windows Metafile and indicates the 16-bit metafile format. This particular record identifies a GDI function call of \textit{Polyline} using points with 16-bit coordinates. You'll find \textit{WmfPolyline} records in metafiles created before the advent of the 32-bit versions of Windows or in metafiles created by 32-bit programs to be backward compatible with earlier code. Such metafiles are still much-used in libraries of clip art.

The \textit{EmfPolyline} member has a prefix of \textit{Emf}, which stands for Enhanced Metafile and indicates the 32-bit metafile format. Again, the record identifies a function call of \textit{Polyline}, but the points have 32-bit coordinates. You'll find such records in enhanced metafiles created by 32-bit versions of Windows.

The \textit{DrawLines} record identifies the \textit{DrawLines} method of the \textit{Graphics} class. You'll find such records only in EMF+ metafiles created by Windows Forms programs.

For backward compatibility with 32-bit Windows programs, the default behavior of the \textit{Metafile} class results in metafiles that actually have two sets of records: EMF records (such as \textit{EmfPolyline}) and EMF+ records (such as \textit{DrawLines}). The EMF records mimic the functionality of the EMF+ records.

However, you can create shorter metafiles if you're using the metafiles in a more restricted way. For example, if you intend the metafiles to be read only by Win32 programs, the metafiles don't need the EMF+ records. If the metafiles will be read only by other Windows Forms programs, the metafiles don't need EMF records.

Here are some simple \textit{Metafile} constructors that have \textit{EmfType} arguments in conjunction with an optional description string:

\textbf{Metafile Constructors (selection)}

\begin{verbatim}
Metafile(string strFileName, IntPtr ipHdc, EmfType et)
Metafile(string strFileName, IntPtr ipHdc, EmfType et, string strDescription)
Metafile(Stream stream, IntPtr ipHdc, EmfType et)
Metafile(Stream stream, IntPtr ipHdc, EmfType et, string strDescription)
Metafile(IntPtr ipHdc, EmfType et)
Metafile(IntPtr ipHdc, EmfType et, string strDescription)
\end{verbatim}

The description string usually describes the image, possibly with a copyright notice. It's embedded in the metafile header. The \textit{EmfType} enumeration is defined like so:

\textbf{EmfType Enumeration}

\begin{verbatim}
<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmfOnly</td>
<td>3</td>
</tr>
<tr>
<td>EmfPlusOnly</td>
<td>4</td>
</tr>
<tr>
<td>EmfPlusDual</td>
<td>5</td>
</tr>
</tbody>
</table>
\end{verbatim}

When you specify an \textit{EmfType} argument of \textit{EmfOnly}, the metafile contains only EMF records. You should use this option if the metafiles your program creates will be used only by non–Windows Forms programs.
When you specify an `EmfType` argument of `EmfPlusOnly`, the metafile contains only EMF+ records. Such metafiles have the advantage of being comparatively small, but they are usable only by other Windows Forms programs.

The `EmfType` argument of `EmfPlusDual` is the default. The metafile contains both EMF and EMF+ records. Consequently, the metafile is just about equal to the size of a corresponding `EmfOnly` metafile plus an `EmfPlusOnly` metafile.

If you want to create a 16-bit WMF from a Windows Forms program, you must create a WMF handle using Win32 code and pass that handle to the appropriate `Metafile` constructor. You can also create a `Metafile` object based on an EMF handle obtained using Win32 code.

So far, I've shown you 22 of the 39 constructors and described the three that create `Metafile` objects from Win32 metafiles handles. The remaining 14 constructors let you specify both a boundary rectangle and a metafile type.

Ten of these constructors begin with a filename or `Stream` followed by an `IntPtr` to a device context handle. Next is either a `Rectangle` or `RectangleF` object with a `MetafileFrameUnit` member. The constructor concludes with an `EmfType` member, a description string, or both. But if the constructor begins with a `Stream`, it must have an `EmfType` member.

The other four constructors begin with an `IntPtr` to a device context handle. Next is a `Rectangle` or `RectangleF` object with a `MetafileFrameUnit` member. The constructor concludes with an `EmfType` member and an optional description string.

Here are the three most generalized, most inclusive, and longest `Metafile` constructors:

**Metafile Constructors (selection)**

| Metafile (string strFileName, IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu, EmfType et, string strDescription) |
| Metafile(Stream stream, IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu, EmfType et, string strDescription) |
| Metafile(IntPtr ipHdc, RectangleF rectf, MetafileFrameUnit mfu, EmfType et, string strDescription) |

**Enumerating the Metafile**

Because metafiles are often stored on disk, the inquisitive programmer can be tempted to open them as regular files and go poking around inside. That sounds like fun to me, but there's also a method of the `Graphics` class that lets you examine metafile records in a more structured manner. Basically, you provide a method in your program that is called for each record in the metafile.

The `EnumerateMetafile` method comes in 36 versions, of which this is the simplest:

**Graphics EnumerateMetafile Methods (selection)**

| void EnumerateMetafile(Metafile mf, Point pt, Graphics.EnumerateMetafileProc emp) |

It may seem odd that `EnumerateMetafile` is a method of our old friend the `Graphics` class. But you'll see shortly that the method not only enumerates a metafile but also provides a way to render it on a record-by-record basis. As your method gets access to every record of the metafile, it can decide to let the record be rendered, skip the record, or (if you're particularly brave) modify the record and render it. And that's also why there are so many versions of the `EnumerateMetafile` method. The methods are similar to the various overloads of `DrawImage`.

Rather than list all 36 versions of `EnumerateMetafile`, here's a summary of the required and optional arguments:
The first argument is always a `Metafile` object.

The second argument is always a destination. You can specify a `Point`, a `PointF`, an array of three `Point` or `PointF` structures, a `Rectangle`, or a `RectangleF`. Just as in `DrawImage`, when you specify an array of three `Point` or `PointF` structures, the points represent the destination of the top left, top right, and bottom left corners of the image.

Next can be an optional argument that indicates a source rectangle within the metafile. If the destination argument is a `Point`, a `Point` array, or a `Rectangle`, the source rectangle must be a `Rectangle`. If the destination argument is a `PointF`, a `PointF` array, or a `RectangleF`, the source rectangle must be a `RectangleF`. The destination rectangle must be followed by a `GraphicsUnit` value indicating the units of the source rectangle.

The next argument is required. It's a method in your program that you've defined in accordance with the `Graphics.EnumerateMetafileProc` delegate. This is the method that gets called for each record in the metafile.

The next argument is optional. It's an `IntPtr` that is defined as a pointer to programmer-defined data that's supposed to be passed to the enumeration method defined in the previous argument. However, there is no argument in the `Graphics.EnumerateMetafileProc` delegate for this programmer-defined data.

If the optional `IntPtr` argument is present, it can be followed by another optional argument, which is an `ImageAttributes` object that determines certain aspects of how the image is displayed.

Here's that simplest call to `EnumerateMetafile` as it might appear in an actual program:

```csharp
grfx.EnumerateMetafile(mf, new Point(0, 0),
    new Graphics.EnumerateMetafileProc(EnumMetafileProc));
```

The last argument makes reference to a method named `EnumMetafileProc` that is defined in accordance with the `Graphics.EnumerateMetafileProc` delegate. Such a method appears in your program looking something like this:

```csharp
bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
    int iDataSize, IntPtr ipData,
    PlayRecordCallback prc)
{
    //
    return bContinue;
}
```

This metafile enumeration method returns `true` to continue enumerating the metafile and `false` otherwise.

The arguments to the enumeration method (which I'll identify briefly here and discuss in more detail shortly) begin with a member of the `EmfPlusRecordType` enumeration that identifies the record. The `iFlags` argument is undocumented.

The `iDataSize` argument indicates the number of bytes that the `ipData` argument points to. This data is unique for each record type. For example, a record type that indicates a polyline would store a point count and multiple points in `ipData`.

The last argument is of type `PlayRecordCallback`, which is a delegate defined in the `System.Drawing.Imaging` namespace. You recall that `EnumerateMetafileProc` is also a delegate. It's very unusual for an argument of a delegate to be another delegate. Here's how the `PlayRecordCallback` delegate is defined:

```csharp
public delegate void PlayRecordCallback(EmfPlusRecordType eprt, int iFlags,
    int iDataSize, IntPtr ipData);
```

Instead of defining a method in your program in accordance with the `PlayRecordCallback` delegate (which is what you usually do with a delegate), the last argument to your enumeration method indicates the method you're supposed to call to render that particular metafile record. The delegate indicates the arguments to that method. So you could define your enumeration method like so:
bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                int iDataSize, IntPtr ipData,
                PlayRecordCallback prc)
{
    prc(eprt, iFlags, iDataSize, ipData);
    return true;
}

Notice the call to the PlayRecordCallback delegate. When you call EnumerateMetafile with this EnumMetafileProc method, EnumerateMetafile should function just like DrawImage, except perhaps a bit slower because EnumMetafileProc is getting access to each metafile record.

Alas, PlayRecordCallback doesn't work, and you should probably ignore the last argument to the enumeration method. Instead, use this method of the Metafile class to render a metafile record in your enumeration method:

Metafile PlayRecord Method

void PlayRecord(EmfPlusRecordType eprt, int iFlags, int iDataSize,
                byte[] abyData)

This method looks a lot like the PlayRecordCallback delegate except that the last argument is an array of bytes instead of an IntPtr. For converting between the IntPtr argument to the enumeration method and the array of bytes required by PlayRecord, you can use the static Copy method in the Marshal class of the System.Runtime.InteropServices namespace.

Because you call PlayRecord in your enumeration method and PlayRecord is a method of the Metafile class, the Metafile object must be stored as a field in your program. Here's an enumeration method that simply renders the metafile:

bool EnumMetafileProc(EmfPlusRecordType eprt, int iFlags,
                int iDataSize, IntPtr ipData,
                PlayRecordCallback prc)
{
    byte[] abyData = new Byte[iDataSize];
    Marshal.Copy(ipData, abyData, 0, iDataSize);
    mf.PlayRecord(eprt, iFlags, iDataSize, abyData);
    return true;
}

The following program creates a write-only text box and a panel control, with a splitter control between them. It also implements an OpenFileDialog dialog box to open a metafile. The metafile is displayed normally on the panel. When the metafile is first loaded, the program creates a StringWriter object and then calls EnumerateMetafile to enumerate the metafile using the EnumMetafileProc method defined in the program. EnumMetafileProc formats the information into the StringWriter object. On return from EnumerateMetafile, the program puts the resultant string into the text box.

EnumMetafile.cs
//-----------------------------------------------------------------------
// EnumMetafile.cs © 2001 by Charles Petzold
//-----------------------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Runtime.InteropServices;
using System.Windows.Forms;

class EnumMetafile : Form
{
    Metafile mf;
    Panel panel;
    TextBox textbox;
    string strCaption;
    StringWriter strwrite;

    public static void Main()
    {
        Application.Run(new EnumMetafile());
    }

    public EnumMetafile()
    {
        Text = strCaption = "Enumerate Metafile";

        // Create the text box for displaying records.
        textbox = new TextBox();
        textbox.Parent = this;
        textbox.Dock = DockStyle.Fill;
        textbox.Multiline = true;
        textbox.WordWrap = false;
        textbox.ReadOnly = true;
        textbox.TabStop = false;
        textbox.ScrollBars = ScrollBars.Vertical;

        // Create the splitter between the panel and the text box.
        Splitter splitter = new Splitter();
        splitter.Parent = this;
        splitter.Dock = DockStyle.Left; // Right;

        // Create the panel for displaying the metafile.
        panel = new Panel();
        panel.Parent = this;
        panel.Dock = DockStyle.Left;
        panel.Paint += new PaintEventHandler(PanelOnPaint);
// Create the menu.

Menu = new MainMenu();
Menu.MenuItems.Add("&Open!", new EventHandler(MenuOpenOnClick));

void MenuOpenOnClikk(object obj, EventArgs ea)
{
    OpenFileDialog dlg = new OpenFileDialog();

    dlg.Filter = "All Metafiles|*.wmf;*.emf|" +
             "Windows Metafile (*\.wmf)|*.wmf|" +
             "Enhanced Metafile (*\emf)|*.emf|";

    if (dlg.ShowDialog() == DialogResult.OK)
    {
        try
        {
            mf = new Metafile(dlg.FileName);
        }
        catch (Exception exc)
        {
            MessageBox.Show(exc.Message, strCaption);
            return;
        }
        Text = strCaption + " - " + Path.GetFileName(dlg.FileName);
        panel.Invalidate();

        // Enumerate the metafile for the text box.

        strwrite = new StringWriter();
        Graphics grfx = CreateGraphics();

        grfx.EnumerateMetafile(mf, new Point(0, 0),
                              new Graphics.EnumerateMetafileProc(EnumMetafileProc));

        grfx.Dispose();
        textbox.Text = strwrite.ToString();
        textbox.SelectionLength = 0;
    }
}
if (iDataSize > 0)
{
    byte[] abyData = new Byte[iDataSize];
    Marshal.Copy(ipData, abyData, 0, iDataSize);

    foreach (byte by in abyData)
        strwrite.Write(" {0:X2}", by);

    strwrite.WriteLine();
    return true;
}

void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel panel = (Panel) obj;
    Graphics grfx = pea.Graphics;

    if (mf != null)
        grfx.DrawImage(mf, 0, 0);
}

Although this program mysteriously doesn't work with any of the metafiles created in this chapter (or with metafiles created by programs from my book Programming Windows), here's the program displaying one of the metafiles included with Visual Studio .NET:

![Image of metafile records]

The metafile records are shown at the right. Following the record type, the flag and the number of bytes of data are in parentheses. The hexadecimal bytes follow. No attempt has been made in this program to convert the bytes into meaningful information.

As you can see, the metafile records are of variable size. Each record type corresponds to a particular GDI function call or a Graphics method call. The data that accompanies each record corresponds to the arguments of the call. Obviously, this data is dependent on the record type.

The new metafile records that correspond to methods of the Graphics class are currently undocumented (at the time I'm writing this) but will undoubtedly be documented sometime in the future. If you're interested in getting a head start on metafile enumeration, you can begin by exploring the format of the older WMF and EMF records. Each record type corresponds to a structure beginning with the letters EMR ("enhanced metafile record") defined in the Win32 header files.

The EMR structure is part of every record. It contains the record type and a size:

typedef struct tagEMR
Here's the structure for a polyline. It begins with an EMR structure.

typedef struct tagEMRPOLYLINE
{
    EMR emr;
    RECTL rclBounds;
    DWORD cptl;
    POINTL aptl[1];
}
EMRPOLYLINE;

Everything following the EMR field corresponds to the data that accompanies the record in the Windows Forms enumeration.

If you're not familiar with Win32 structures, be aware that RECTL and POINTL have long fields (but those are 32-bit C long integers, not 64-bit C# long integers), and RECTS and POINTS structures have short fields, which are 16-bits in length. Moreover, a rectangle is defined in terms of the upper left corner and lower right corner, not in terms of the upper left corner and the width and height.
Overview

The Windows clipboard allows data to be transferred from one application to another. The clipboard is a relatively simple mechanism that doesn't require much overhead in either the program that places data on it or the program that later gets access to it. Most applications use the Windows clipboard even when cut-and-paste operations involve the transfer of data solely within the application.

Programs that deal with documents or other data give the user access to the clipboard through the standard menu options Cut, Copy, and Paste, duplicated by the keyboard shortcuts (inherited from the Apple Macintosh) Ctrl+X, Ctrl+C, and Ctrl+V. When a user selects Cut or Copy, the program transfers selected data from the application to the clipboard. The data is in a particular format or formats, usually text, bitmap, metafile, or binary. The Cut command has the additional effect of deleting the selection from the document. When a user selects Paste from the menu, the program determines whether the clipboard contains data in a format that the program can use. If it does, the program transfers the data from the clipboard to the program.

Programs should not transfer data to the clipboard without an explicit instruction from the user. For example, a user who performs a Cut or Copy (or a Ctrl+X or Ctrl+C) operation in one program should be able to assume that the data will remain on the clipboard until the next Cut or Copy operation.

As you may recall from Chapter 18, the TextBox and RichTextBox controls implement their own clipboard interfaces. A program using these controls need only call the appropriate methods implemented in TextBoxBase. In the general case, however, you don't have that convenience. You must instead access the clipboard yourself.

The drag-and-drop facility in Windows is closely related to the clipboard, so I'll be discussing that in this chapter as well.

Items and Formats

Only one item is stored on the clipboard at any time. Whenever a program copies an item to the clipboard, the new item replaces what was there before.

However, an application can copy a particular item to the clipboard in multiple formats. For example, consider a spreadsheet program in which the user selects an array of rows and columns, and then triggers the Copy command. The spreadsheet program will probably use a variety of formats for storing those rows and columns on the clipboard. Of most importance to the application is probably a binary format known only to the application itself; this private format allows the program to retrieve the exact data (with any formatting that is present) when the user triggers Paste.

The spreadsheet program can also copy the selected rows and columns into the clipboard in a text-based comma-separated format that other spreadsheet or database programs can use. It could also format the data into text using tabs between the columns; this format is suitable when you paste the data into a word processing program. Perhaps the spreadsheet application could even create a bitmap or a metafile containing an image of the rows and columns.

The application pasting data from the clipboard doesn't need to handle all these different formats. It can simply select the format it can use most efficiently.

As you might expect, the existence of clipboard items in multiple formats adds a layer of complexity to clipboard programming.

The Tiny (But Powerful) Clipboard Class

The Clipboard class is part of the System.Windows.Forms namespace. The class is sealed and has no public constructors. You can't instantiate it, and you can't inherit from it. The class has no properties and just two static methods, one of which comes in two versions. You use these methods to set data to the clipboard or get data out:

Clipboard Static Methods
void SetDataObject(object obj)
void SetDataObject(object obj, bool bRemainAfterExit)
IDataObject GetDataObject()

As long as you're using only one data format, putting data on the clipboard is the easier of the two operations. For example, if you have a string named str that you want to copy to the clipboard, simply call
Clipboard.SetDataObject(str, true);

Whatever was on the clipboard before is deleted and replaced with this item. Similarly, you can put a bitmap on the clipboard:
Clipboard.SetDataObject(bitmap, true);

You can also put a metafile on the clipboard:
Clipboard.SetDataObject(metafile, true);

In all three of these cases, a copy is made of the object for the clipboard's purposes. It's OK if you change the object after the SetDataObject call. It won't change what's on the clipboard. This sequence of statements won't cause any problems:
Clipboard.SetDataObject(str, true);
str = null;

Once something is on the clipboard, the only way you can affect it is with another call to SetDataObject; that second call to SetDataObject replaces the clipboard item with a new one.

I've set the second argument of SetDataObject to true in these examples because that's probably what you'll want to use whenever you put a string, a bitmap, or a metafile on the clipboard. If you set the second argument to false,
Clipboard.SetDataObject(str, false);
or if you don't include the argument at all,
Clipboard.SetDataObject(str);

the item that you put on the clipboard disappears when your program terminates. It's probably best for the user if the item is still on the clipboard regardless of whether or not your program is still running.

However, the items you put on the clipboard are not limited to strings, bitmaps, and metafiles. You can put any object on the clipboard. Here's some code that creates a new Button object, sets the Text property, and then copies the object to the clipboard:
Button btn = new Button();
btn.Text = "OK";
Clipboard.SetDataObject(btn);

In cases where the object passed to SetDataObject is not a string, a metafile, or a bitmap, you must use the short form of SetDataObject or specify false as the second argument. The reason for this restriction is that the clipboard can't be used for transferring arbitrary objects (such as Button objects) between applications. Only the application that put the Button object on the clipboard can retrieve it. Thus, it makes no sense for the object to be on the clipboard after the program terminates. Like I said, putting data on the clipboard is easiest if you restrict yourself to one format. Later in this chapter, I'll demonstrate how you can put multiple data formats on the clipboard.

Getting data from the clipboard isn't quite as simple as putting data on the clipboard. The GetDataObject method is defined as returning an instance of a class that implements the IDataObject interface, which means that you call GetDataObject like so:
IDataObject data = Clipboard.GetDataObject()
The object called `data` now contains everything you need to know about the contents of the clipboard, and our attention must now focus on the `IDataObject` interface.

**Getting Objects from the Clipboard**

I'm first going to show you a fairly simple and straightforward way to get objects from the clipboard, a job you'll probably perform in response to the user selecting Paste from the menu.

The static `GetDataObject` method of the `Clipboard` class isn't documented as returning an object of a particular class. It's documented only as returning an instance of a class that implements the `IDataObject` interface. That gives us enough information to use the methods defined in `IDataObject` using the object returned from `Clipboard.GetDataObject`. `IDataObject` defines four methods (12 if you count overloads). Here are two of them in their simplest forms:

**`IDataObject` Methods (selection)**

```csharp
bool GetDataPresent(Type type)
object GetData(Type type)
```

The `GetDataPresent` method should probably be named `IsDataPresent` to be more consistent with the rest of the Windows Forms methods and properties. If you have an object named `data` returned from `Clipboard.GetDataObject`, the expression

```csharp
data.GetDataPresent(typeof(String))
```

returns `true` if the clipboard contains a `String` object. Notice the use of the `typeof` operator to get the `Type` object that identifies the `String` class. Similarly,

```csharp
data.GetDataPresent(typeof(Bitmap))
```

returns `true` if there's a `Bitmap` object on the clipboard, and

```csharp
data.GetDataPresent(typeof(Bitmap))
```

returns `true` if a `Metafile` object is available. If you're in the habit of putting nonstandard objects on the clipboard, you can also make calls like this:

```csharp
data.GetDataPresent(typeof(Button))
```

It's possible to make these calls without actually saving the return value from the `Clipboard.GetDataObject` call. For example,

```csharp
Clipboard.GetDataObject().GetDataPresent(typeof(Bitmap))
```

returns `true` if the clipboard contains a `Bitmap` object.

It's customary to use the `GetDataPresent` method during the `Popup` event of the Edit menu. You enable the Paste item only if `GetDataPresent` returns `true` for the data type you're interested in.

By the way, the fact that

```csharp
data.GetDataPresent(typeof(String))
```

returns `true` doesn't imply that `GetDataPresent` will return `false` for other types. Like I said, the clipboard can contain multiple formats of the same clipboard item. The clipboard could contain a `String` object with some text, a `Metafile` object containing a `DrawString` call displaying that same text string, and a `Bitmap` rendition of the same text. The application getting data from the clipboard should check for the most useful format.

To get an object from the clipboard, you call the `GetData` method. For example,

```csharp
string str = (string) data.GetData(typeof(string));
```

Because `GetData` returns an `object`, the return value must be cast to the desired data type. Similarly,

```csharp
Bitmap bitmap = (Bitmap) data.GetData(typeof(Bitmap));
```
Getting an object from the clipboard doesn't affect the contents of the clipboard. The return value of `GetData` is a copy of the object stored on the clipboard.

It's time to see how this all works in actual code. The following program does little more than copy `String` objects to and from the clipboard.

**ClipText.cs**

```csharp
//---------------------------------------
// ClipText.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ClipText : Form
{
    string strText = "Sample text for the clipboard";
    MenuItem miCut, miCopy, miPaste;

    public static void Main()
    {
        Application.Run(new ClipText());
    }
    public ClipText()
    {
        Text = "Clip Text";
        ResizeRedraw = true;

        Menu = new MainMenu();

        // Edit menu

        MenuItem mi = new MenuItem("&Edit");
        mi.Popup += new EventHandler(MenuEditOnPopup);
        Menu.MenuItems.Add(mi);

        // Edit Cut menu item

        miCut = new MenuItem("Cu&t");
        miCut.Click += new EventHandler(MenuEditCutOnClick);
        miCut.Shortcut = Shortcut.CtrlX;
        Menu.MenuItems[0].MenuItems.Add(miCut);

        // Edit Copy menu item

        miCopy = new MenuItem("&Copy");
        miCopy.Click += new EventHandler(MenuEditCopyOnClick);
        

miCopyShortcut = Shortcut.CtrlC;
Menu.MenuItems[0].MenuItems.Add(miCopy);

    // Edit Paste menu item

miPaste = new MenuItem("&Paste");
miPaste.Click += new EventHandler(MenuEditPasteOnClick);
miPaste.Shortcut = Shortcut.CtrlV;
Menu.MenuItems[0].MenuItems.Add(miPaste);
void MenuEditOnPopup(object obj, EventArgs ea)
{
    miCut.Enabled =
    miCopy.Enabled = strText.Length > 0;
    miPaste.Enabled =
    Clipboard.GetDataObject().GetDataPresent(typeof(string));
}
void MenuEditCutOnClick(object obj, EventArgs ea)
{
    MenuEditCopyOnClick(obj, ea);
    strText = "";
    Invalidate();
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    Clipboard.SetDataObject(strText, true);
}
void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();
    if (data.GetDataPresent(typeof(string)))
    strText = (string) data.GetData(typeof(string));
    Invalidate();
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;
    StringFormat strfmt = new StringFormat();
    strfmt.Alignment = strfmt.LineAlignment =
StringAlignment.Center;
    grfx.DrawString(strText, Font, new SolidBrush(ForeColor),
    ClientRectangle, strfmt);
The ClipText program maintains a string variable named strText that it displays centered in its client area. The constructor creates an Edit menu with Cut, Copy, and Paste items. The Popup event handler enables the Cut and Copy items only if the string has a nonzero length. The Paste item is enabled only if the clipboard contains a string object.

The Click event handler for the Copy command uses the SetDataObject method of Clipboard to copy strText to the clipboard. The Cut event handler calls the Copy event handler and also deletes the string from the program by setting strText to the empty string.

The Click event handler for the Paste command first checks whether the clipboard still contains an object of type string. (You might find that check redundant considering that the Paste item isn't enabled if the clipboard isn't storing text. However, given that Windows is a multitasking environment, it's possible for the clipboard contents to change between the time a submenu is displayed and an item is clicked. Calling GetData for an object type that no longer exists on the clipboard won't cause an exception to be raised, but GetData will return a null value, and ClipText isn't quite prepared for that eventuality.) If the clipboard contains text, the Paste event handler calls GetData to obtain the string object and then assigns that string to strText.

You can experiment with ClipText in conjunction with Microsoft Notepad, word processors, and Web browsers. As you'll see, when you copy text from a word processor or a Web browser and paste it into ClipText, the text loses any formatting it had. That result isn't unexpected: you know that string objects normally don't include any formatting, and you'd probably be startled to see ClipText display a block of text with rich text format (RTF) or HTML tags. I'll explain shortly how you can go beyond plain text with the clipboard.

In Chapter 23, I introduced the MetafileViewer program and the MetafileConvert program, which inherited from MetafileViewer. The following program inherits from MetafileConvert and implements an Edit menu to transfer metafiles to and from the clipboard.

MetafileClip.cs

//-------------------------------
// MetafileClip.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

class MetafileClip: MetafileConvert
{
    MenuItem miCut, miCopy, miPaste, miDel;

    public new static void Main()
    {
        Application.Run(new MetafileClip());
    }
    public MetafileClip()
    {
        Text = strProgName = "Metafile Clip";

        // Edit menu
Menu.MenuItems[1].Popup += new EventHandler(MenuEditOnPopup);

    // Edit Cut menu item
    miCut = new MenuItem("Cut");
    miCut.Click += new EventHandler(MenuEditCutOnClick);
    miCut.Shortcut = Shortcut.CtrlX;
    Menu.MenuItems[1].MenuItems.Add(miCut);

    // Edit Copy menu item
    miCopy = new MenuItem("Copy");
    miCopy.Click += new EventHandler(MenuEditCopyOnClick);
    miCopy.Shortcut = Shortcut.CtrlC;
    Menu.MenuItems[1].MenuItems.Add(miCopy);

    // Edit Paste menu item
    miPaste = new MenuItem("Paste");
    miPaste.Click += new EventHandler(MenuEditPasteOnClick);
    miPaste.Shortcut = Shortcut.CtrlV;
    Menu.MenuItems[1].MenuItems.Add(miPaste);

    // Edit Delete menu item
    miDel = new MenuItem("Delete");
    miDel.Click += new EventHandler(MenuEditDelOnClick);
    miDel.Shortcut = Shortcut.Del;
    Menu.MenuItems[1].MenuItems.Add(miDel);
} void MenuEditOnPopup(object obj, EventArgs ea)
{
    miCut.Enabled =
    miCopy.Enabled =
    miDel.Enabled = mf != null;
    miPaste.Enabled =
        Clipboard.GetDataObject().GetDataPresent(typeof(Metafile));
} void MenuEditCutOnClick(object obj, EventArgs ea)
{
    MenuEditCopyOnClick(obj, ea);
    MenuEditDelOnClick(obj, ea);
} void MenuEditCopyOnClick(object obj, EventArgs ea)
{
}
Clipboard.SetDataObject(mf, true);
}

void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();
    if (data.GetDataPresent(typeof(Metafile)))
        mf = (Metafile) data.GetData(typeof(Metafile));

    strFileName = "clipboard";
    Text = strProgName + " - " + strFileName;
    Invalidate();
}

void MenuEditDelOnClick(object obj, EventArgs ea)
{
    mf = null;
    strFileName = null;
    Text = strProgName;
    Invalidate();
}

Besides Cut, Copy, and Paste, this program also includes a Delete item on the Edit menu. (In some
applications, an item named Clear is essentially the equivalent of Delete.) The Delete option doesn't
actually involve the clipboard because (unlike Cut) it deletes without first copying to the clipboard.
However, if you're already implementing Cut and Copy, adding Delete is usually trivial. In fact, you
can think of a Cut operation as a Copy followed by a Delete. That's exactly how the Click event
handler for the Cut option is written:
void MenuEditCutOnClick(object obj, EventArgs ea)
{
    MenuEditCopyOnClick(obj, ea);
    MenuEditDelOnClick(obj, ea);
}

Because the MetafileClip program deals with files as well as the clipboard, the other complication
involves dealing with the form's caption bar. In the earlier versions of the program, I set the Text
property of the form to the program name (separated into words) stored in the strProgName field
along with the currently loaded filename stored in the strFileName field:
Metafile Viewer - Picture.emf
In the MetafileClip version, the strProgName text is "Metafile Clip".

When the Delete option is selected, the strFileName variable must be set to null and Text set to just
the strProgName. That much is obvious. The more difficult problem is what should be done when a
metafile is loaded from the clipboard. I decided to set the strFileName field to "clipboard". Other
possibilities are "untitled" or "metafile".

In Chapter 16, "Dialog Boxes," I introduced the ImageOpen program to load Image objects from files
and the ImageIO program to save them. In Chapter 21, "Printing," the ImagePrint program added
printing capabilities. Now the ImageClip program inherits from ImagePrint to add clipboard capability.
ImageClip.cs
//----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

class ImageClip: ImagePrint
{
    MenuItem miCut, miCopy, miPaste, miDel;

    public new static void Main()
    {
        Application.Run(new ImageClip());
    }
    public ImageClip()
    {
        Text = strProgName = "Image Clip";

        // Edit menu
        MenuItem mi = new MenuItem("&Edit");
        mi.Popup += new EventHandler(MenuEditOnPopup);
        Menu.MenuItems.Add(mi);
        int index = Menu.MenuItems.Count - 1;

        // Edit Cut menu item
        miCut = new MenuItem("Cu&t");
        miCut.Click += new EventHandler(MenuEditCutOnClick);
        miCut.Shortcut = Shortcut.CtrlX;
        Menu.MenuItems[index].MenuItems.Add(miCut);

        // Edit Copy menu item
        miCopy = new MenuItem("&Copy");
        miCopy.Click += new EventHandler(MenuEditCopyOnClick);
        miCopy.Shortcut = Shortcut.CtrlC;
        Menu.MenuItems[index].MenuItems.Add(miCopy);

        // Edit Paste menu item
        miPaste = new MenuItem("&Paste");
        miPaste.Click += new EventHandler(MenuEditPasteOnClick);
        miPaste.Shortcut = Shortcut.CtrlV;
Menu.MenuItems[index].MenuItems.Add(miPaste);

    // Edit Delete menu item

    miDel = new MenuItem("Delete");
    miDel.Click += new EventHandler(MenuEditDelOnClick);
    miDel.Shortcut = Shortcut.Del;
    Menu.MenuItems[index].MenuItems.Add(miDel);
}
void MenuEditOnPopup(object obj, EventArgs ea)
{
    miCut.Enabled =
    miCopy.Enabled =
    miDel.Enabled = image != null;

    IDataObject data = Clipboard.GetDataObject();

    miPaste.Enabled = data.GetDataPresent(typeof(Bitmap)) ||
    data.GetDataPresent(typeof(Metafile));
}
void MenuEditCutOnClick(object obj, EventArgs ea)
{
    MenuEditCopyOnClick(obj, ea);
    MenuEditDelOnClick(obj, ea);
}
void MenuEditCopyOnClick(object obj, EventArgs ea)
{
    Clipboard.SetDataObject(image, true);
}
void MenuEditPasteOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(typeof(Metafile)))
        image = (Image) data.GetData(typeof(Metafile));
    else if (data.GetDataPresent(typeof(Bitmap)))
        image = (Image) data.GetData(typeof(Bitmap));

    strFileName = "Clipboard";
    Text = strProgName + " - " + strFileName;
    Invalidate();
}
void MenuEditDelOnClick(object obj, EventArgs ea)
{
The enhancements here are similar to those in MetafileClip, but the ImageClip program adds another complexity. It's actually dealing with an `Image` object (stored as the field `image`), which can be either a `Bitmap` or a `Metafile` object. There's no problem with using an `Image` object in the `SetDataObject` method:

```csharp
Clipboard.SetDataObject(image, true);
```

The `SetDataObject` method probably calls `GetType` on the first argument to determine the type of the object. Depending on what type the `image` object is, `GetType` will return `System.Drawing.Bitmap` or `System.Drawing.Imaging.Metafile`.

However, you can't use `typeof(Image)` in the `GetDataPresent` or `GetData` method of the `IDataObject` interface. If your program has the ability to paste either a bitmap or a metafile, you should enable the Paste menu item if the clipboard contains either a bitmap or a metafile:

```csharp
miPaste.Enabled = data.GetDataPresent(typeof(Bitmap)) ||
    data.GetDataPresent(typeof(Metafile));
```

It's possible that the clipboard contains both a `Bitmap` and a `Metafile` representing the same image. When the user selects Paste, the program has to decide which one to load.

I decided that the `Metafile` object should take priority:

```csharp
if (data.GetDataPresent(typeof(Metafile)))
    image = (Image) data.GetData(typeof(Metafile));
else if (data.GetDataPresent(typeof(Bitmap)))
    image = (Image) data.GetData(typeof(Bitmap));
```

This was not an arbitrary choice! Ask yourself, What kind of program puts both a metafile and a bitmap on the clipboard? It's probably a drawing program—a program that essentially works with metafiles. When such an application copies an image to the clipboard, it also converts the image to a bitmap. That way the image is available to applications that can't handle metafiles.

It's highly unlikely that the metafile originated with a paint program. Paint programs don't usually have the ability to convert bitmaps to metafiles. Such conversions are rather specialized and work only with very simple images.

Thus, the metafile is the real image and the bitmap is only a conversion. A program that can deal with both metafiles and bitmaps should load metafiles from the clipboard in preference to bitmaps.

**Clipboard Data Formats**

At first, it seems so simple: you put an object of type `String`, `Bitmap`, `Metafile`, or even `Button` on the clipboard, and you extract an object of type `String`, `Bitmap`, `Metafile`, or `Button` from the clipboard.

But not every application running under Windows is a Windows Forms program! Some Windows programs place objects on the clipboard that don't directly correspond to Windows Forms types. The problem goes both ways: Windows applications that make use of the Win32 API or MFC are not prepared to deal directly with Windows Forms objects.

Let's examine the seemingly simple data type known as `text`. Some existing Windows programs store text in Unicode, but most of them don't. If a Windows program passes 8-bit ANSI text strings to the clipboard, a Windows Forms program should be able to read the text as a normal `String` object. Likewise, when a Windows Forms program puts a `String` object on the clipboard, it should be
readable by programs that know only the ANSI character set. There's even another text encoding known as OEM text that dates back to the character-mode environment of MS-DOS. OEM stands for "original equipment manufacturer," but in the United States, it really refers to the 8-bit character set IBM used in the original PC. (You may remember the line-drawing characters used by character-mode programs.) ANSI text and OEM text differ in the upper 128 characters.

These requirements suggest that the clipboard must perform conversions among multiple text encodings. Regardless of the encoding of the text that goes on the clipboard, the clipboard must make several additional encodings available to other applications.

It's already fairly clear that these conversions between different encodings are taking place. The ClipText program successfully transfers text to and from regular Windows programs that use either ANSI or Unicode text encoding.

However, the problems of text don't stop with character encodings. Consider a word processor that copies a block of text to the clipboard. What happens to the formatting of the text? If the user wants to paste such text back into the same word processing program (perhaps in the same document or in a different one), the text certainly shouldn't lose its formatting. On the other hand, if the user wants to paste the text into Notepad, all the formatting should be stripped from the text. What should happen if you paste the text into another word processing program? You probably want the formatting to be preserved, and that implies that the text on the clipboard should be stored in a way that includes formatting in an application-independent manner (RTF, for example, or possibly HTML).

Or suppose you copy text from a Web browser to the clipboard. The HTML formatting should be preserved for applications that understand HTML, but it shouldn't be imposed on applications that can't deal with it. Most users probably don't want to see HTML tags when they copy text from a Web browser to Notepad!

Some text is intended to be read by programs rather than people. For example, many database and spreadsheet programs can copy and paste information in a comma-separated value (CSV) text format. This format provides an application-independent way of sharing database records or numeric tables.

So, the simple statement that copies a text string to the clipboard,

```
Clipboard.SetDataObject(str);
```

probably doesn't quite do what you want if `str` is a text string that consists of RTF, HTML, or CSV text. Preferably there should be some way for an application to specify what type of text it's actually putting on the clipboard.

When a word processing application copies text to the clipboard, the application itself is probably in the best position to convert its internal format into RTF and plain, unformatted text. The implication is that applications should be able to set data on the clipboard in several different formats at once. (At first, that doesn't seem possible. When you call `SetDataObject`, the object passed as an argument replaces the current contents of the clipboard. You can't call `SetDataObject` multiple times to put multiple formats of the same text on the clipboard. However, as we'll soon see, the argument to `SetDataObject` can be an object that itself specifies several different objects.)

To a Windows Forms program, bitmaps and metafiles are objects of type `Bitmap` and `Metafile`. To a non–Windows Forms program, however, there are two types of bitmaps: the old-style, device-dependent bitmaps that were introduced in Windows 1.0 and the DIBs (device-independent bitmaps) introduced in Windows 3.0. And, as I discussed in Chapter 23, non–Windows Forms programs deal with the original metafile format (WMF) and the enhanced metafile format (EMF).

Here's the key to making all this work: the clipboard stores not only chunks of data but also identifications of the data formats. To a Windows Forms program, a particular clipboard format is usually identified by a text string, such as "DeviceIndependentBitmap" or "Text". These text strings are also associated with ID numbers, such as 8 for "DeviceIndependentBitmap" and 1 for "Text". The ID numbers are the same as the identifiers beginning with `CF` ("clipboard format") defined in the Win32 header files.

The `GetFormats` method defined in the `IDataObject` interface provides a list of all the data formats currently stored on the clipboard:
**IDataObject GetFormats Methods**

string[] GetFormats()

string[] GetFormats(bool bIncludeConversions)

The call

```csharp
string[] astr = data.GetFormats();
```

is equivalent to

```csharp
string[] astr = data.GetFormats(true);
```

Both calls return string identifications of all the clipboard formats currently available from the clipboard. Some of these formats represent conversions from the data currently stored on the clipboard. To restrict the list to just unconverted formats, use

```csharp
string[] astr = data.GetFormats(false);
```

Sometimes data is converted as it's put on the clipboard. Those formats will be returned regardless of the `GetFormats` argument.

For example, if a Windows Forms program puts a `String` object on the clipboard, `GetFormats` (regardless of the argument) returns the strings

System.String
UnicodeText
Text

The "UnicodeText" and "Text" types allow a Win32 API or MFC program to obtain the clipboard text in either Unicode or an 8-bit ANSI encoding.

When Notepad puts text on the clipboard, `GetFormats` with a `false` argument returns these four strings:

UnicodeText
Locale
Text
OEMText

A call to `GetFormats` with no arguments or a `true` argument returns those four strings and this one as well:

System.String

When a Windows Forms program puts a `Bitmap` object on the clipboard, then regardless of the arguments, `GetFormats` returns the strings

System.Drawing.Bitmap
WindowsForms10PersistentObject
Bitmap

When a Windows Forms program puts a `Metafile` object on the clipboard, then regardless of the arguments, `GetFormats` returns the strings

System.Drawing.Imaging.Metafile
WindowsForms10PersistentObject

It's also possible to use the format strings in the `GetDataPresent` and `GetData` methods defined in the `IDataObject` interface:

**IDataObject Methods (selection)**
bool GetDataPresent(string strFormat)
bool GetDataPresent(string strFormat, bool bAllowConversions)
object GetData(string strFormat)
object GetData(string strFormat, bool bAllowConversions)

The call
string str = (string) data.GetData(typeof(string))

is equivalent to
string str = (string) data.GetData("System.String");

That's pretty obvious. But those two calls are also equivalent to this one:
string str = (string) data.GetData("UnicodeText");

The *GetData* method always converts the clipboard data into a .NET Framework data type, so in this case *GetData* returns a *string* object.

Consider the following call:

string str = (string) data.GetData("Text");

Is this one equivalent as well? Maybe, maybe not. If a Unicode-aware program put a string on the clipboard that contained Hebrew, Arabic, or Cyrillic characters, for example, the string identified with "Text" is an 8-bit ANSI version of the original Unicode text. The Unicode characters that have no equivalents in the ANSI character set are replaced by question marks.

In any event, if you set the second argument of *GetDataPresent* or *GetData* to *false*, you won't get a converted type. The call

(string) data.GetData("Text", false)

returns *null* if Unicode text was put on the clipboard.

If you prefer not to hard-code those text strings in your program, you can make use of the *DataFormats* class. This class contains 21 static read-only fields that return the text strings for those clipboard formats directly supported by the .NET Framework. Here are the clipboard formats that originated in the Win32 API. The column at the right shows the Win32 clipboard ID number associated with each format.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Value</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Text</td>
<td>&quot;Text&quot;</td>
<td>1</td>
</tr>
<tr>
<td>string</td>
<td>Bitmap</td>
<td>&quot;Bitmap&quot;</td>
<td>2</td>
</tr>
<tr>
<td>string</td>
<td>MetafilePict</td>
<td>&quot;MetaFilePict&quot;</td>
<td>3</td>
</tr>
<tr>
<td>string</td>
<td>SymbolicLink</td>
<td>&quot;SymbolicLink&quot;</td>
<td>4</td>
</tr>
<tr>
<td>string</td>
<td>Dif</td>
<td>&quot;DataInterchangeFormat&quot;</td>
<td>5</td>
</tr>
<tr>
<td>string</td>
<td>Tiff</td>
<td>&quot;TaggedImageFileFormat&quot;</td>
<td>6</td>
</tr>
<tr>
<td>string</td>
<td>OemText</td>
<td>&quot;OEMText&quot;</td>
<td>7</td>
</tr>
<tr>
<td>string</td>
<td>Dib</td>
<td>&quot;DeviceIndependentBitmap&quot;</td>
<td>8</td>
</tr>
<tr>
<td>string</td>
<td>Palette</td>
<td>&quot;Palette&quot;</td>
<td>9</td>
</tr>
<tr>
<td>string</td>
<td>PenData</td>
<td>&quot;PenData&quot;</td>
<td>10</td>
</tr>
</tbody>
</table>
### DataFormats Static Fields (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Value</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Riff</td>
<td>&quot;RiffAudio&quot;</td>
<td>11</td>
</tr>
<tr>
<td>string</td>
<td>WaveAudio</td>
<td>&quot;WaveAudio&quot;</td>
<td>12</td>
</tr>
<tr>
<td>string</td>
<td>UnicodeText</td>
<td>&quot;UnicodeText&quot;</td>
<td>13</td>
</tr>
<tr>
<td>string</td>
<td>EnhancedMetafile</td>
<td>&quot;EnhancedMetafile&quot;</td>
<td>14</td>
</tr>
<tr>
<td>string</td>
<td>FileDrop</td>
<td>&quot;FileDrop&quot;</td>
<td>15</td>
</tr>
<tr>
<td>string</td>
<td>Locale</td>
<td>&quot;Locale&quot;</td>
<td>16</td>
</tr>
</tbody>
</table>

For example, the expression

```
DataFormats.Text
```

returns the string "Text", and the expression

```
DataFormats.Dib
```

returns the string "DeviceIndependentBitmap". I'll be referring to these formats by their text names because the text versions most accurately reflect the manner in which the clipboard identifies the data format.

The "Text" and "OEMText" formats are both 8-bit encodings. The "Text" format is the ANSI encoding used by most Windows programs. The "OEMText" encoding is the character set used in character-mode MS-DOS programs. The "OEMText" format is provided so that you can copy, cut, and paste in the MS-DOS Command Prompt window. A clipboard item identified by the format "Locale" is a number that usually accompanies 8-bit character encodings to identify the character set. The "UnicodeText" format identifies Unicode text.

The "SymbolicLink" string identifies text in the Symbolic Link (SYLK) format created by Microsoft for the MultiPlan spreadsheet program, and the "DataInterchangeFormat" string identifies text in the Data Interchange Format (DIF) devised by Software Arts for the VisiCalc spreadsheet program. Both these clipboard formats were introduced in Windows 1.0 and, as you might expect, aren't used much these days.

When a Windows Forms program specifies a clipboard format of "Text", "OEMText", "UnicodeText", "SymbolicLink", or "DataInterchangeFormat" in the `GetData` method, the method returns an object of type `String`.

The "Bitmap" and "DeviceIndependentBitmap" strings identify the device-dependent and the device-independent bitmaps, respectively. The "Palette" format identifies a color palette format used in conjunction with 256-color DIBs. The "TaggedImageFileFormat" string identifies the TIFF bitmap format.

The "MetaFilePict" and "EnhancedMetafile" strings represent the old and enhanced metafile formats, respectively.

When a Windows Forms program specifies a clipboard format of "Bitmap" or "DeviceIndependentBitmap" in the `GetData` method, the method returns an object of type `Bitmap`. For "Enhanced Metafile", `GetData` returns an object of type `Metafile`. However, for "MetaFilePict", `GetData` returns an object of type `MemoryStream`. That's simply a block of memory that a program can access using `ReadByte` and `Read` methods defined by the `MemoryStream` class. `GetData` returns a `MemoryStream` object for "MetaFilePict" because old-style metafiles are not stored in the clipboard directly. Instead, the clipboard stores a small C structure that references a handle to the metafile.

The "PenData" string is used in conjunction with the (now abandoned) pen extensions to Windows. "RiffAudio" identifies multimedia data in the Resource Interchange File Format, and "WaveAudio" identifies waveform audio files. The first release of Windows Forms has no multimedia support. For these clipboard formats (and for "Palette"), `GetData` returns an object of type `MemoryStream`. 
Finally, "FileDrop" identifies a list of files that probably originated in Windows Explorer. (Select one or more files, and then perform a Copy or Cut to place a "FileDrop" object on the clipboard.) To a Windows Forms program, a "FileDrop" item is an array of strings. Although you can use "FileDrop" items with the clipboard, the format is used more often in conjunction with drag-and-drop operations (which I'll discuss toward the end of this chapter).

So far, most of these additional formats are not very useful to the Windows Forms programmer. When a Windows Forms program puts a `String` object on the clipboard, the `String` object is automatically converted to the "Text" and "UnicodeText" formats for other applications. If another application puts text (of whatever type) on the clipboard, that text is automatically converted to a `String` object for a Windows Forms program.

Similarly, as we've seen, Windows Forms programs can transfer bitmaps and metafiles to and from the clipboard without getting involved with the conversions necessary for non–Windows Forms programs.

Five additional static fields are defined in the `DataFormats` class that are of more use to Windows Forms programs:

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>CommaSeparatedValue</td>
<td>&quot;Csv&quot;</td>
</tr>
<tr>
<td><code>string</code></td>
<td>Html</td>
<td>&quot;HTML Format&quot;</td>
</tr>
<tr>
<td><code>string</code></td>
<td>Rtf</td>
<td>&quot;Rich Text Format&quot;</td>
</tr>
<tr>
<td><code>string</code></td>
<td>Serializable</td>
<td>&quot;WindowsForms10PersistentObject&quot;</td>
</tr>
<tr>
<td><code>string</code></td>
<td>StringFormat</td>
<td>&quot;System.String&quot;</td>
</tr>
</tbody>
</table>

These formats are not currently defined in the Win32 clipboard interface, but some Win32 programs use the first three.

"Csv" is a text format used by spreadsheet and database programs for exchanging data. In a block of spreadsheet rows and columns, columns are separated by commas and rows are separated by end-of-line characters. When a database uses "Csv", fields are separated by commas and records are separated by end-of-line characters. Numbers are stored in a readable ASCII format; text is delimited by quotation marks.

Although "Csv" is a text format, a call to `GetData` with a "Csv" argument does not return an object of type `string`. It returns a `MemoryStream` object that contains null-terminated text. To extract the `string` object from this `MemoryStream` object, you can use the following code:

```csharp
MemoryStream memorystream = (MemoryStream) data.GetData("Csv");
StreamReader streamreader = new StreamReader(memorystream);
string str = streamreader.ReadToEnd();
```

You'll also need a `using` statement for the `System.IO` namespace. (See Appendix A for more information about the `MemoryStream` and `StreamReader` classes.) The `string` object will probably end with a "\x0000" character.

When a Windows Forms program specifies "HTML Format" or "Rich Text Format", the `GetData` method returns an object of type `string`. However, in the former case, the `string` object will contain HTML tags along with the text. In the latter case, the `string` object may or may not contain RTF tags. Because plain text is a subset of RTF, the `string` object could contain just plain text.

The "WindowsForms10PersistentObject" format shows up when a Windows Forms program has copied a `Bitmap` or `Metafile` object to the clipboard. The return type of `GetData` is either `Bitmap` or `Metafile`. You don't generally need to use this format.

As I mentioned earlier, the "System.String" format causes `GetData` to return an object of type `String`. It's exactly like using an argument of `typeof(String)`. 
Here's a program that expands its Paste menu to include options to paste plain text, RTF, HTML, or CSV.

**RichTextPaste.cs**

```csharp
//-------------------------------
// RichTextPaste.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class RichTextPaste: Form
{
    string strPastedText = "";
    MenuItem miPastePlain, miPasteRTF, miPasteHTML, miPasteCSV;

    public static void Main()
    {
        Application.Run(new RichTextPaste());
    }
    public RichTextPaste()
    {
        Text = "Rich-Text Paste";
        ResizeRedraw = true;

        Menu = new MainMenu();

        // Edit menu
        MenuItem mi = new MenuItem("&Edit");
        mi.Popup += new EventHandler(MenuEditOnPopup);
        Menu.MenuItems.Add(mi);

        // Edit Paste Plain Text menu item
        miPastePlain = new MenuItem("Paste &Plain Text");
        miPastePlain.Click += new EventHandler(MenuEditPastePlainOnClick);
        Menu.MenuItems[0].MenuItems.Add(miPastePlain);

        // Edit Paste RTF menu item
        miPasteRTF = new MenuItem("Paste &Rich Text Format");
        miPasteRTF.Click += new EventHandler(MenuEditPasteRTFOnClick);
        Menu.MenuItems[0].MenuItems.Add(miPasteRTF);
```
// Edit Paste HTML menu item

miPasteHTML = new MenuItem("Paste &HTML");
miPasteHTML.Click += new EventHandler(MenuEditPasteHTMLOnClick);
Menu.MenuItems[0].MenuItems.Add(miPasteHTML);

// Edit Paste CSV menu item

miPasteCSV = new MenuItem("Paste &Comma-Separated Values");
miPasteCSV.Click += new EventHandler(MenuEditPasteCSVOnClick);
Menu.MenuItems[0].MenuItems.Add(miPasteCSV);
}

void MenuEditOnPopup(object obj, EventArgs ea)
{
    miPastePlain.Enabled =
        Clipboard.GetDataObject().GetDataPresent(typeof(string));
    miPasteRTF.Enabled =
        Clipboard.GetDataObject().GetDataPresent(DataFormats.Rtf);
    miPasteHTML.Enabled =
        Clipboard.GetDataObject().GetDataPresent(DataFormats.Html);
    miPasteCSV.Enabled =
        Clipboard.GetDataObject().GetDataPresent(DataFormats.CommaSeparatedValue);
}

void MenuEditPastePlainOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(typeof(string)))
    {
        strPastedText = (string) data.GetData(typeof(string));
        Invalidate();
    }
}

void MenuEditPasteRTFOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(DataFormats.Rtf))
    {
        strPastedText = (string) data.GetData(DataFormats.Rtf);
        Invalidate();
    }
}

void MenuEditPasteHTMLOnClick(object obj, EventArgs ea)
Try selecting some text in a Web browser, copying it to the clipboard, and then using this program to see what formats are available. For CSV, the code to convert the `MemoryStream` object to a `string` is similar to the code I showed earlier. For RTF and HTML, the program simply displays the text without attempting to parse the formatting tags.

### Clipboard Viewers

A clipboard viewer is a program that displays the current contents of the clipboard. Here’s a clipboard viewer that contains 21 radio buttons corresponding to the 21 fields of the `DataFormats` class. The program sets a 1-second timer and checks the clipboard contents during the `Tick` event. (The Win32 messages that inform an application when the clipboard contents have changed are not directly available to a Windows Forms program.) The radio buttons are enabled according to what formats are available. When you click a button, the clipboard item in that format is rendered on the right side of the form.

`ClipView.cs`

```csharp
(productId) = Clipboard.GetDataObject();

if (productId.GetDataPresent(DataFormats.Html))
{
    strPastedText = (string) productId.GetData(DataFormats.Html);
    Invalidate();
}
}
void MenuEditPasteCSVOnClick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    if (data.GetDataPresent(DataFormats.CommaSeparatedValue))
    {
        MemoryStream memstr = (MemoryStream) data.GetData(" Csv");
        StreamReader streamreader = new StreamReader(memstr);
        strPastedText = streamreader.ReadToEnd();
        Invalidate();
    }
}
protected override void OnPaint(PaintEventArgs pea)
{
    Graphics grfx = pea.Graphics;

    grfx.DrawString(strPastedText, Font, new SolidBrush(ForeColor),
            ClientRectangle);
}
```
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class ClipView: Form
{
    string[] astrFormats =
    {
        DataFormats.Bitmap, DataFormats.CommaSeparatedValue, DataFormats.Dib,
        DataFormats.Dif, DataFormats.EnhancedMetafile, DataFormats.FileDrop,
        DataFormats.Html, DataFormats.Locale, DataFormats.MetafilePict,
        DataFormats.OemText, DataFormats.Palette, DataFormats.PenData,
        DataFormats.Riff, DataFormats.Rtf, DataFormats.Serializable,
        DataFormats.StringFormat, DataFormats.SymbolicLink, DataFormats.Text,
        DataFormats.Tiff, DataFormats.UnicodeText, DataFormats.WaveAudio
    };

    Panel panelDisplay;
    RadioButton[] aradio;
    RadioButton radioChecked;

    public static void Main()
    {
        Application.Run(new ClipView());
    }

    public ClipView()
    {
        Text = "Clipboard Viewer";

        // Create variable-width panel for clipboard display.
        panelDisplay = new Panel();
        panelDisplay.Parent = this;
        panelDisplay.Dock = DockStyle.Fill;
        panelDisplay.Paint += new PaintEventHandler(PanelOnPaint);
        panelDisplay.BorderStyle = BorderStyle.Fixed3D;

        // Create splitter.
        Splitter split = new Splitter();
        split.Parent = this;
        split.Dock = DockStyle.Left;

        // Create panel for radio buttons.
    }
}
Panel panel = new Panel();
panel.Parent = this;
panel.Dock = DockStyle.Left;
panel.Width = 200;

    // Create radio buttons.

aradio = new RadioButton[astrFormats.Length];
EventHandler eh = new EventHandler(RadioButtonOnClick);

for (int i = 0; i < astrFormats.Length; i++)
{
    aradio[i] = new RadioButton();
aradio[i].Parent = panel;
aradio[i].Location = new Point(4, 12 * i);
aradio[i].Size = new Size(300, 12);
aradio[i].Click += eh;
aradio[i].Tag = astrFormats[i];
}
    // Set autoscale base size.

AutoScaleBaseSize = new Size(4, 8);

    // Set time for 1 second.

Timer timer = new Timer();
timer.Interval = 1000;
timer.Tick += new EventHandler(TimerOnTick);
timer.Enabled = true;
}
void TimerOnTick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    for (int i = 0; i < astrFormats.Length; i++)
    {
        aradio[i].Text = astrFormats[i];
aradio[i].Enabled = data.GetDataPresent(astrFormats[i]);

        if (aradio[i].Enabled)
        {
            if (!data.GetDataPresent(astrFormats[i], false))
aradio[i].Text += "*";
        }
        object objClip = data.GetData(astrFormats[i]);
    }
try
{
    aradio[i].Text += " (" + objClip.GetType() + ")";
}
catch
{
    aradio[i].Text += " (Exception on GetType!)";
}
}
panelDisplay.Invalidate();
}
void RadioButtonOnClick(object obj, EventArgs ea)
{
    radioChecked = (RadioButton) obj;
    panelDisplay.Invalidate();
}
void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel     panel = (Panel) obj;
    Graphics  grfx  = pea.Graphics;
    Brush     brush = new SolidBrush(panel.ForeColor);

    if (radioChecked == null || !radioChecked.Enabled)
        return;

    IDataObject data = Clipboard.GetDataObject();

    object objClip = data.GetData((string) radioChecked.Tag);

    if (objClip == null)
        return;

    else if (objClip.GetType() == typeof(string))
    {
        grfx.DrawString((string)objClip, Font, brush,
                        panel.ClientRectangle);
    }
    else if (objClip.GetType() == typeof(string[]))   // FileDrop
    {
        string str = string.Join("\r\n", (string[]) objClip);
        grfx.DrawString(str, Font, brush, panel.ClientRectangle);
    }
else if (objClip.GetType() == typeof(Bitmap) ||
objClip.GetType() == typeof(Metafile) ||
objClip.GetType() == typeof(Image))
{
    grfx.DrawImage((Image)objClip, 0, 0);
}
else if (objClip.GetType() == typeof(MemoryStream))
{
    Stream stream = (Stream) objClip;
    byte[] abyBuffer = new byte[16];
    long   lAddress = 0;
    int    iCount;
    Font   font = new Font(FontFamily.GenericMonospace,
                           Font.SizeInPoints);
    float  y = 0;

    while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0)
    {
        string str = HexDump.ComposeLine(lAddress, abyBuffer,
                                          iCount);
        grfx.DrawString(str, font, brush, 0, y);
        lAddress += 16;
        y += font.GetHeight(grfx);

        if (y > panel.Bottom)
            break;
    }
}
}

During the **Tick** event handler, the button text is set to the text version of the clipboard format. If the format is available, the button text also includes an asterisk if the format has been converted from another clipboard format. The .NET data type of the clipboard format is enclosed in parentheses.

Here's what the program looks like after I've copied part of the program text from Visual C# .NET to the clipboard:
The `PanelOnPaint` method is responsible for updating the panel at the right. It can handle several .NET data types. For `string`, the text is simply displayed using `DrawText`. `DrawText` is also used for an array of `string` objects, which is the case for the "FileDrop" clipboard type. For data types of `Bitmap` and `Metafile`, the `PanelOnPaint` method uses `DrawImage`. And for the data type of `MemoryStream`, `PanelOnPaint` uses the static `ComposeLine` method from the HexDump program in Appendix A.

The ClipView program doesn't list every format of data on the clipboard. It shows only those formats that are directly supported within Windows Forms by virtue of being represented by a field in the `DataFormats` class. As I mentioned earlier, it's possible to use the `GetFormats` method defined by the `IDataObject` interface to get a `string` array of all the formats of the current clipboard item. That's what the ClipViewAll program uses.

**ClipViewAll.cs**

```csharp
//------------------------------------------
// ClipViewAll.cs © 2001 by Charles Petzold
//------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;

class ClipViewAll: Form
{
    Panel       panelDisplay, panelButtons;
    RadioButton radioChecked;
    string[]    astrFormatsSave = new string[0];

    public static void Main()
    {
        Application.Run(new ClipViewAll());
    }
    public ClipViewAll()
    {
        Text = "Clipboard Viewer (All Formats)";
    }
}
```
// Create variable-width panel for clipboard display.
panelDisplay = new Panel();
panelDisplay.Parent = this;
panelDisplay.Dock = DockStyle.Fill;
panelDisplay.Paint += new PaintEventHandler(PanelOnPaint);
panelDisplay.BorderStyle = BorderStyle.Fixed3D;

// Create splitter.
Splitter split = new Splitter();
split.Parent = this;
split.Dock = DockStyle.Left;

// Create panel for radio buttons.
panelButtons = new Panel();
panelButtons.Parent = this;
panelButtons.Dock = DockStyle.Left;
panelButtons:autoScroll = true;
panelButtons.Width = Width / 2;

// Set time for 1 second.
Timer timer = new Timer();
timer.Interval = 1000;
timer.Tick += new EventHandler(TimerOnTick);
timer.Enabled = true;
}
void TimerOnTick(object obj, EventArgs ea)
{
    IDataObject data = Clipboard.GetDataObject();

    string[] astrFormats = data.GetFormats();
    bool bUpdate = false;

    // Determine whether clipboard formats have changed.
    if (astrFormats.Length != astrFormatsSave.Length)
    {
        bUpdate = true;
    }
    else
    {
        for (int i = 0; i < astrFormats.Length; i++)
        {
            if (astrFormats[i] != astrFormatsSave[i])
            ...
        }
{
    bUpdate = true;
    break;
}

// Invalidate display regardless.

panelDisplay.Invalidate();

// Don't update buttons if formats haven't changed.

if (!bUpdate)
    return;

// Formats have changed, so re-create radio buttons.

astrFormatsSave = astrFormats;
panelButtons.Controls.Clear();
Graphics grfx = CreateGraphics();
EventHandler eh = new EventHandler(RadioButtonOnClick);
int cxText = AutoScaleBaseSize.Width;
int cyText = AutoScaleBaseSize.Height;

for (int i = 0; i < astrFormats.Length; i++)
{
    RadioButton radio = new RadioButton();
    radio.Parent = panelButtons;
    radio.Text = astrFormats[i];
    if (!data.GetDataPresent(astrFormats[i], false))
        radio.Text += "*";
    try
    {
        object objClip = data.GetData(astrFormats[i]);
        radio.Text += " (" + objClip.GetType() + ")";
    }
    catch
    {
        radio.Text += " (Exception on GetData or GetType!)";
    }
    radio.Tag = astrFormats[i];
    radio.Location = new Point(cxText, i * 3 * cyText / 2);
    radio.Size = new Size((radio.Text.Length + 20) * cxText, 3 * cyText / 2);
radio.Click += eh;
}
grfx.Dispose();
radioChecked = null;
}
void RadioButtonOnClick(object obj, EventArgs ea)
{
    radioChecked = (RadioButton) obj;
    panelDisplay.Invalidate();
}
void PanelOnPaint(object obj, PaintEventArgs pea)
{
    Panel    panel = (Panel) obj;
    Graphics grfx  = pea.Graphics;
    Brush    brush = new SolidBrush(panel.ForeColor);

    if (radioChecked == null)
        return;

    IDataObject data = Clipboard.GetDataObject();
    object objClip = data.GetData((string) radioChecked.Tag);

    if (objClip == null)
        return;

    else if (objClip.GetType() == typeof(string))
    {
        grfx.DrawString((string)objClip, Font, brush,
                        panel.ClientRectangle);
    }
    else if (objClip.GetType() == typeof(string[]))   // FileDrop
    {
        string str = string.Join("\r\n", (string[]) objClip);

        grfx.DrawString(str, Font, brush, panel.ClientRectangle);
    }
    else if (objClip.GetType() == typeof(Bitmap) ||
             objClip.GetType() == typeof(Metafile) ||
             objClip.GetType() == typeof(Image))
    {
        grfx.DrawImage((Image)objClip, 0, 0);
    }
    else if (objClip.GetType() == typeof(MemoryStream))
    {
        Stream stream = (Stream) objClip;
byte[] abyBuffer = new byte[16];
long lAddress = 0;
int iCount;
Font font = new Font(FontFamily.GenericMonospace,
                      Font.SizeInPoints);
float y = 0;

while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0)
{
    string str = HexDump.ComposeLine(lAddress, abyBuffer,
                                        iCount);
    grfx.DrawString(str, font, brush, 0, y);
    lAddress += 16;
    y += font.GetHeight(grfx);

    if (y > panel.Bottom)
        break;
}

Every second, this program checks whether the clipboard formats have changed and, if they have, re-creates a collection of radio buttons, one for each format. Like the ClipView program, the text for each radio button also indicates whether the format is native and the .NET type of the data.

Here's what the program looks like after I've copied some text from Microsoft Word to the clipboard:

Yes, this display does indeed reveal that the clipboard contains 13 different formats of the same text item.

Because the clipboard provides a medium for applications to exchange data, it's essential that you test your clipboard code with other applications. You'll find the ClipView and ClipViewAll programs useful for exploring the clipboard from the perspective of a Windows Forms program, but the data your program copies to the clipboard must also make sense to non–Windows Forms programs.

If you begin exploring the various clipboard formats that some applications use (such as the list in the ClipViewAll display from Word), you'll find that some of them originated in the OLE (object linking and embedding) specification, while others (like "HyperlinkWordBkmk") are obviously private to the application.

Using a private clipboard format is simple. You just make up a name for the format, store it in a string, and use that string as the clipboard format. You'll want to avoid collisions with other applications using their own private formats, so give the format a name you're sure will be unique. It's
pretty easy if you use the name of the application as part of the clipboard format name, for example, "WriteALot Version 2.1 Formatted Text".

Win32 programs generally refer to clipboard formats using ID numbers. (The first table of the DataFormats fields earlier in this chapter shows the ID numbers associated with the standard clipboard formats.) Windows also assigns identification numbers when applications use nonstandard formats. The DataFormats class has a static method named GetFormat that essentially translates the format name and the format ID:

**DataFormats Static Methods**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Format GetFormat(int id)</code></td>
</tr>
<tr>
<td><code>Format GetFormat(string strName)</code></td>
</tr>
</tbody>
</table>

Format is another class defined within DataFormats, so it appears in the class library documentation as DataFormats.Format. The class has just two, read-only properties:

**Format Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Id</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>Name</td>
<td>get</td>
</tr>
</tbody>
</table>

For example, the call

```csharp
DataFormats.GetFormat("DeviceIndependentBitmap").Id
```

returns 8, and the call

```csharp
DataFormats.GetFormat(8).Name
```

returns the string "DeviceIndependentBitmap". Nonstandard formats return numbers that can vary from session to session, so you shouldn't hard-code them in your programs.

If you use a private clipboard format, you should supplement it with standard formats. That requires that you use multiple clipboard formats for the same clipboard item.

**Setting Multiple Clipboard Formats**

As you've seen, you use the static SetDataObject method of the Clipboard class to put data on the clipboard. Here's a call that puts text on the clipboard:

```csharp
Clipboard.SetDataObject(strText);
```

You use similar calls for putting objects of type Bitmap or Metafile on the clipboard.

But what if the `strText` variable contains a block of RTF or HTML text? How do you indicate that fact in the SetDataObject call? And what if you need to put multiple formats on the clipboard? Because each call to SetDataObject replaces the item already on the clipboard, that doesn't seem possible.

The solution to both these problems is the DataObject class. Recall that the static GetDataObject method of Clipboard is documented as returning an object of a class that implements the IObjectWithConnection interface. DataObject is the only class in the .NET Framework that implements the IObjectWithConnection interface. (This doesn't necessarily mean that Clipboard.GetDataObject returns an object of type DataObject. It could create a new class dynamically. But you'll find that Clipboard.GetDataObject often does return an object of type DataObject.)

DataObject has three constructors:

**DataObject Constructors**

```
DataObject()
```
DataObject(object objData)
DataObject(string strFormat, object objData)

For example, the call
Clipboard.SetDataObject(strText);

is equivalent to
Clipboard.SetDataObject(new DataObject(strText));

or
Clipboard.SetDataObject(new DataObject(DataFormats.StringFormat, strText));

If strText actually contains a block of HTML, you can use
Clipboard.SetDataObject(new DataObject(DataFormats.Html, strText));

if the item shouldn't remain on the clipboard after the program terminates, and
Clipboard.SetDataObject(new DataObject(DataFormats.Html, strText), true);

if it should remain.

DataObject implements the IDataObject interface, so it supports all the methods defined for
IDataObject; the class doesn't support any other methods or properties. I've already discussed the
GetDataPresent, GetData, and GetFormats methods. The only remaining method is SetData, and
that's the one you'll use when you use DataObject for defining multiple formats:

DataObject SetData Method

void SetData(object objData)
void SetData(Type typeFormat, object objData)
void SetData(string strFormat, object objData)
void SetData(string strFormat, bool bConvert, object objData)

You create an object of type DataObject, use calls to SetData to store multiple formats of a single
item, and then pass the DataObject object to Clipboard.SetDataObject. By default, items are
converted to compatible formats unless you use the last overload in the table and set the bConvert
argument to false.

For example, suppose strText is a string of plain text you want to copy to the clipboard, strHtml
contains the same text with HTML formatting, and strRtf is the same string with RTF formatting.
Here's a sequence for storing these multiple formats on the clipboard:

DataObject data = new DataObject();
data.SetData(strText);
data.SetData(DataFormats.Html, strHtml);
data.SetData(DataFormats.Rtf, strRtf);
Clipboard.SetDataObject(data, true);

Watch out: Nobody's checking whether strHtml and strRtf are really blocks of HTML and RTF!
Likewise, any data that your program obtains from the clipboard might not necessarily be what it's
labeled. You'll want to parse HTML and RTF text strings from the clipboard very carefully.
Here's a program, MultiCopy, that copies a two-dimensional array of float values (defined as a field named `afValues`) to the clipboard in three different formats: a private format, the CSV format, and a plain text format.

**MultiCopy.cs**

```csharp
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;

class MultiCopy : Form
{
    const string strFormat = "MultiCopy.InternalFormat";

    float[,] afValues = {{ 0.12f, 3.45f, 6.78f, 9.01f },
                          { 2.34f, 5.67f, 8.90f, 1.23f },
                          { 4.56f, 7.89f, 0.12f, 3.45f }};

    public static void Main()
    {
        Application.Run(new MultiCopy());
    }

    public MultiCopy()
    {
        Text = "Multi Copy";
        Menu = new MainMenu();

        // Edit menu
        MenuItem mi = new MenuItem("&Edit");
        Menu.MenuItems.Add(mi);

        // Edit Copy menu item
        mi = new MenuItem("&Copy");
        mi.Click += new EventHandler(MenuEditCopyOnClick);
        mi.Shortcut = Shortcut.CtrlC;
        Menu.MenuItems[0].MenuItems.Add(mi);
    }

    void MenuEditCopyOnClick(object obj, EventArgs ea)
    {
        DataObject data = new DataObject();

        // Define internal clipboard format.
    }
}
```
MemoryStream memorystream = new MemoryStream();
BinaryWriter binarywriter = new BinaryWriter(memorystream);

binarywriter.Write(afValues.GetLength(0));
binarywriter.Write(afValues.GetLength(1));

for (int iRow = 0; iRow < afValues.GetLength(0); iRow++)
    for (int iCol = 0; iCol < afValues.GetLength(1); iCol++)
        binarywriter.Write(afValues[iRow, iCol]);

binarywriter.Close();

data.SetData(strFormat, memorystream);

    // Define CSV clipboard format.

memorystream = new MemoryStream();
StreamWriter streamwriter = new StreamWriter(memorystream);

for (int iRow = 0; iRow < afValues.GetLength(0); iRow++)
    for (int iCol = 0; iCol < afValues.GetLength(1); iCol++)
        {
            streamwriter.Write(afValues[iRow, iCol]);

            if (iCol < afValues.GetLength(1) - 1)
                streamwriter.Write(",");
            else
                streamwriter.WriteLine();
        }
streamwriter.Write("\0");
streamwriter.Close();
data.SetData(DataFormats.CommaSeparatedValue, memorystream);

    // Define String clipboard format.

StringWriter stringwriter = new StringWriter();

for (int iRow = 0; iRow < afValues.GetLength(0); iRow++)
    for (int iCol = 0; iCol < afValues.GetLength(1); iCol++)
        {
            stringwriter.Write(afValues[iRow, iCol]);

            if (iCol < afValues.GetLength(1) - 1)
                stringwriter.Write("\t");
        }
else
    StringWriter.WriteLine();
}
stringWriter.Close();
data.SetData(stringWriter.ToString());

Clipboard.SetDataObject(data, false);
}
}

Most of what this program does takes place during the MenuEditCopyOnClick event handler. That method begins by defining an object of type DataObject and concludes by calling Clipboard.SetDataObject to copy the object to the clipboard. In between, the data is formatted in three different ways.

A private format is identified by the string "MultiCopy.InternalFormat". The array of float values (preceded by the integer number of rows and columns) is stored in a binary format in a MemoryStream object. The method uses the BinaryWriter class (discussed in Appendix A) to facilitate the writing of binary objects to the stream. The method adds the memory stream to the DataObject object using the call

data.SetData(strFormat, memorystream);

where strFormat is the string "MultiCopy.InternalFormat".

Next the method formats the data in CSV. Again, it creates a MemoryStream object for storing the data, and this time it creates a StreamWriter object to facilitate the addition of formatted text strings to the stream. Values in the same row are separated by commas; lines are separated by carriage returns and line feeds. The method adds this memory stream to the DataObject object using the call

data.SetData(DataFormats.CommaSeparatedValue, memorystream);

Finally, the information is formatted into plain text. The process is much like CSV formatting except that tabs are used rather than commas to separate values in the same row. Another difference is that the text isn't put into a memory stream. Instead, the program uses a StringWriter object to construct a string containing formatted text. This format is added to the DataObject object using the call

data.SetData(stringWriter.ToString());

Drag and Drop

The drag-and-drop facility in Windows allows a user to grab something with the mouse and drag it to another part of the same application or a different application. Usually what the user grabs is one or more files or a block of text, although images and other types of data can also be dragged and dropped.

The application from which you drag an object is called the drag-and-drop source. The application that you drag the object to is the drag-and-drop target. Drag-and-drop usually requires that the left mouse button be pressed, although some applications allow dragging with the right button pressed.

The Windows Explorer application is very often a drag-and-drop source for a list of files (referred to as the clipboard type "FileDrop"). For example, if you select a file in Windows Explorer and then drag it to the client area of Notepad, Notepad will open the file and display it. The Microsoft WordPad program can be a drag-and-drop source and target for text. If you select some text in WordPad, you can drag it to another application that serves as a drag-and-drop target. Similarly, WordPad can be a target for dragged text.

Data dragged from one application to another is generally moved, copied, or linked, depending on the status of the Shift and Ctrl keys:

<table>
<thead>
<tr>
<th>Drag-and-Drop Actions</th>
<th>Key Pressed</th>
<th>Action</th>
</tr>
</thead>
</table>

Drag-and-Drop Actions

<table>
<thead>
<tr>
<th>Key Pressed</th>
<th>Action</th>
</tr>
</thead>
</table>

Drag-and-Drop Actions
Drag-and-Drop Actions

<table>
<thead>
<tr>
<th>Key Pressed</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Move</td>
</tr>
<tr>
<td>Ctrl</td>
<td>Copy</td>
</tr>
<tr>
<td>Shift+Ctrl</td>
<td>Link</td>
</tr>
</tbody>
</table>

In a Move operation, the drag-and-drop source deletes the object. In a Copy, the target receives a copy of the object. In a Link, the source and target each get references to the same object.

If you use the mouse to grab a file list in Windows Explorer or a block of text in WordPad, and you drag that to the client area of any program shown in this book so far, the cursor will change to a circle with a slash—the international no-no sign. To become a drag-and-drop target, a control or form must first have its `AllowDrop` property set to `true`:

**Control Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>AllowDrop</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The following four events are associated with being a drag-and-drop target:

**Control Events (selection)**

<table>
<thead>
<tr>
<th>Event</th>
<th>Method</th>
<th>Delegate</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DragEnter</td>
<td>OnDragEnter</td>
<td>DragEventHandler</td>
<td>DragEventArgs</td>
</tr>
<tr>
<td>DragOver</td>
<td>OnDragOver</td>
<td>DragEventHandler</td>
<td>DragEventArgs</td>
</tr>
<tr>
<td>DragDrop</td>
<td>OnDragDrop</td>
<td>DragEventHandler</td>
<td>DragEventArgs</td>
</tr>
<tr>
<td>DragLeave</td>
<td>OnDragLeave</td>
<td>EventHandler</td>
<td>EventArgs</td>
</tr>
</tbody>
</table>

A `DragEnter` event occurs when a control or form has its `AllowDrop` property set to `true` and the mouse pointer dragging an object first enters the control or the form's client area. After that `DragEnter` event, as the mouse is moved within the control or the client area, `DragOver` events occur. If the mouse is then moved outside the control or the client area, a `DragLeave` event occurs.

A control or form can signal its receptiveness to being a target for the dragged data during the `DragEnter` event or during one of the subsequent `DragOver` events. (We’ll see how shortly.) At that point, the cursor changes from a slashed circle to an arrow with a little box on its tail, possibly accompanied by a plus sign (for a Copy) or a curved arrow (for a Link). If the mouse is then released over the client area, a `DragDrop` event occurs.

If the control or form doesn't signal its receptiveness to the data, the cursor remains a slashed circle. If the mouse is released over the client area, a `DragLeave` event occurs, not a `DragDrop`.

Generally, you'll want to handle the `DragOver` and `DragDrop` events. During the `DragOver` event, you decide whether you can accept the data that's being dragged to your control or form. If only certain areas of the control or form are valid for a drop, you can signal when the drop is valid and when it's not. During the `DragDrop` event, you actually get access to the data. (It's just like a clipboard paste.)

The `DragEnter`, `DragOver`, and `DragDrop` events are all accompanied by an object of type `DragEventArgs`, which has the following properties:

**DragEventArgs Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>KeyState</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>X</td>
<td>get</td>
</tr>
<tr>
<td>int</td>
<td>Y</td>
<td>get</td>
</tr>
</tbody>
</table>
DragEventArgs Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDataObject</td>
<td>Data</td>
<td>get</td>
</tr>
<tr>
<td>DragDropEffects</td>
<td>AllowedEffect</td>
<td>get</td>
</tr>
<tr>
<td>DragDropEffects</td>
<td>Effect</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The first three properties give you some information about the keyboard and mouse at the time of the event. The KeyState property uses a set of bit flags to indicate which mouse buttons and modifier keys are currently pressed:

**KeyState Bit Flags**

<table>
<thead>
<tr>
<th>Key or Button</th>
<th>Bit Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left mouse button</td>
<td>0x01</td>
</tr>
<tr>
<td>Right mouse button</td>
<td>0x02</td>
</tr>
<tr>
<td>Shift key</td>
<td>0x04</td>
</tr>
<tr>
<td>Ctrl key</td>
<td>0x08</td>
</tr>
<tr>
<td>Middle mouse button</td>
<td>0x10</td>
</tr>
<tr>
<td>Alt key</td>
<td>0x20</td>
</tr>
</tbody>
</table>

The X and Y properties indicate the location of the mouse pointer in screen coordinates. (Use the PointToClient method of Control to convert to client area coordinates.)

The next property is named Data, and it's an object of a class that implements the IDataObject interface, just as in the GetDataObject method of the Clipboard class. During the DragEnter or DragOver event, you can use the GetFormats or GetDataPresent methods to determine whether the data is of a type your program can handle. During the DragDrop event, you use the GetData method to obtain a copy of the data.

The AllowedEffect property contains one or more members of the DragDropEffects enumeration:

**DragDropEffects Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Copy</td>
<td>0x00000001</td>
</tr>
<tr>
<td>Move</td>
<td>0x00000002</td>
</tr>
<tr>
<td>Link</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Scroll</td>
<td>0x80000000</td>
</tr>
<tr>
<td>All</td>
<td>0x80000003</td>
</tr>
</tbody>
</table>

The AllowedEffect property is effectively set by the drag-and-drop source to indicate the options available to a drag-and-drop target. Most commonly, a drag-and-drop source will set AllowedEffect to DragDropEffects.Copy | DragDropEffects.Move | DragDropEffects.Link

During the DragEnter and DragOver events, and based on the KeyState, X, Y, Data, and AllowedEffect properties of the DragEventArgs object, the potential drag-and-drop target decides whether it can accept the dropped data. If it can, it sets the Effect property to one of the members of the DragDropEffects enumeration, a member that is included in the AllowedEffect property. Generally, the target uses the KeyState property to determine whether Effect should be set to the Copy, Move, or Link member. (That's what controls the appearance of the cursor.) Setting the Effect property to DragDropEffects.None signals that the target can't accept the data. After the user drops
the object, the drag-and-drop source is informed which member of the enumeration the target
specified.

For any particular drag-and-drop operation, the potential drag-and-drop target needs to set the _Effect_
property only once; that value will be reflected in the _DragEventArgs_ argument of subsequent
_DragOver_ and _DragDrop_ events. It's as if a single _DragEventArgs_ object were used for the entire
drag-and-drop operation. However, the _DragOver_ event handler will probably want to change _Effect_
based on the current status of the Ctrl and Shift keys.

In **Chapter 18**, "Edit, List, and Spin," I began a series of programs that progressively attempted to
emulate Notepad. The adventure continued in **Chapter 21**, "Printing," and is now about to come to a
conclusion. The following file makes the program a drag-and-drop target for text or a "FileDrop" list.
(The real Notepad is only a target for "FileDrop"). The file is called simply NotepadClone.cs because
the program is as finished as it's going to get in this book.

**NotepadClone.cs**

```csharp
// NotepadClone.cs © 2001 by Charles Petzold

using System;
using System.Drawing;
using System.Windows.Forms;

class NotepadClone: NotepadCloneWithPrinting
{
    public new static void Main()
    {
        // This needs to be done for drag-and-drop to work.

        System.Threading.ApartmentState.STA;

        Application.Run(new NotepadClone());
    }

    public NotepadClone()
    {
        strProgName = "NotepadClone";
        MakeCaption();

        textbox.AllowDrop = true;
        textbox.DragOver += new DragEventHandler(TextboxOnDragOver);
        textbox.DragDrop += new DragEventHandler(TextboxOnDragDrop);
    }

    void TextboxOnDragOver(object obj, DragEventArgs dea)
    {
        if (dea.Data.GetDataPresent(DataFormats.FileDrop) ||
            dea.Data.GetDataPresent(DataFormats.StringFormat))
        {
            if ((dea.AllowedEffect & DragDropEffects.Move) != 0)
```
dea.Effect = DragDropEffects.Move;

if (((dea.AllowedEffect & DragDropEffects.Copy) != 0) &&
((dea.KeyState & 0x08) != 0)) // Ctrl key
    dea.Effect = DragDropEffects.Copy;
}
}

void TextBoxOnDragDrop(object obj, DragEventArgs dea)
{
    if (dea.Data.GetDataPresent(DataFormats.FileDrop))
    {
        if (!OkToTrash())
            return;

        string[] astr = (string[])
            dea.Data.GetData(DataFormats.FileDrop);
        LoadFile(astr[0]); // In NotepadCloneWithFile.cs
    }
    else if (dea.Data.GetDataPresent(DataFormats.StringFormat))
    {
        textbox.SelectedText =
            (string)
            dea.Data.GetData(DataFormats.StringFormat);
    }
}
}

During the constructor, the program sets the AllowDrop property of the TextBox control to true and
sets handlers for the text box’s DragOver and DragDrop events. During the DragOver event, the
program checks for data formats of “FileDrop” or “System.String” and then sets the Effect property of
the DragEventArgs object to either DragDropEffects.Move or DragDropEffects.Copy, depending on
what the drag-and-drop source supports and the status of the Ctrl key.

During the DragDrop event, the program does something different, depending on the data format. A
format of “FileDrop” causes the program to load the file. Although “FileDrop” usually indicates a list
of files, NotepadClone can use only one file. For a format of “System.String”, the program performs
an operation similar to a Paste.

You can experiment with transferring text into NotepadClone by using the program in conjunction
with a drag-and-drop source such as WordPad. As you drag something from WordPad to
NotepadClone, you can control the appearance of the cursor by pressing and releasing the Ctrl key.
When the DragDrop event finally occurs, WordPad is notified of the last setting of the Effect property.
WordPad is responsible for deleting or not deleting the selected text.

Let’s now examine what’s involved in becoming a drag-and-drop source. Any class descended from
Control can initiate a drag-and-drop operation by calling the following method, generally in response
to a MouseDown event:

**Control Methods (selection)**

DragDropEffects DoDragDrop(object objData, DragDropEffects dde)
The first argument is the object that the drag-and-drop source has to offer. This argument could be an object of type \texttt{DataObject} if the drag-and-drop source can provide data in multiple formats or if it wants to be more explicit about the format of the data (for example, specifying “HTML Format” for a \texttt{string} type). The second argument is one or more members of the \texttt{DragDropEffects} enumeration.

The method doesn't return until the drag-and-drop operation has completed. At that point, \texttt{DoDragDrop} returns a member of the \texttt{DragDropEffects} enumeration specified by the drag-and-drop target or \texttt{DragDropEffects.None} if the target didn't accept the data or the operation was aborted in some way.

Although \texttt{DoDragDrop} doesn't return until the operation has completed, a control or form can be periodically notified during the process by handling the \texttt{QueryContinueDrag} event. In the following table, ellipses are used to indicate the event name in the method, delegate, and event argument names:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Event & Method & Delegate & Argument \\
\hline
\texttt{QueryContinueDrag} & On… & …\texttt{EventHandler} & …\texttt{EventArgs} \\
\hline
\end{tabular}
\end{table}

The \texttt{QueryContinueDragEventArgs} object that accompanies the event has the following properties:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Type & Property & Accessibility \\
\hline
\texttt{int} & \texttt{KeyState} & get \\
\texttt{bool} & \texttt{EscapePressed} & get \\
\texttt{DragAction} & \texttt{Action} & get/set \\
\hline
\end{tabular}
\end{table}

The drag-and-drop source can set the \texttt{Action} property to one of the following members of the \texttt{DragAction} enumeration:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Member & Value \\
\hline
\texttt{Continue} & 0 \\
\texttt{Drop} & 1 \\
\texttt{Cancel} & 2 \\
\hline
\end{tabular}
\end{table}

Normally, the drag-and-drop operation will be cancelled if the user presses the Esc key. You can override that behavior—or cancel the operation for other reasons—by handling this event.

Here's a program that overrides and enhances ImageClip to become both a drag-and-drop source and target.

\begin{verbatim}
ImageDrop.cs
//--------------------------------------------------------
// ImageDrop.cs © 2001 by Charles Petzold
//--------------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;
\end{verbatim}
class ImageDrop: ImageClip
{
    bool bIsTarget;

    public new static void Main()
    {
        Application.Run(new ImageDrop());
    }

    public ImageDrop()
    {
        Text = strProgName = "Image Drop";
        AllowDrop = true;
    }

    protected override void OnDragOver(DragEventArgs dea)
    {
        if (dea.Data.GetDataPresent(DataFormats.FileDrop) ||
            dea.Data.GetDataPresent(typeof(Metafile)) ||
            dea.Data.GetDataPresent(typeof(Bitmap)))
        {
            if ((dea.AllowedEffect & DragDropEffects.Move) != 0)
                dea.Effect = DragDropEffects.Move;

            if (((dea.AllowedEffect & DragDropEffects.Copy) != 0) &&
                ((dea.KeyState & 0x08) != 0))  // Ctrl key
                dea.Effect = DragDropEffects.Copy;
        }
    }

    protected override void OnDragDrop(DragEventArgs dea)
    {
        if (dea.Data.GetDataPresent(DataFormats.FileDrop))
        {
            string[] astr = (string[]) dea.Data.GetData(DataFormats.FileDrop);
            try
            {
                image = Image.FromFile(astr[0]);
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message, Text);
                return;
            }
            strFileName = astr[0];
            Text = strProgName + " - " + Path.GetFileName(strFileName);
Invalidate();
} else {
    if (dea.Data.GetDataPresent(typeof(Metafile)))
        image = (Image) dea.Data.GetData(typeof(Metafile));
    else if (dea.Data.GetDataPresent(typeof(Bitmap)))
        image = (Image) dea.Data.GetData(typeof(Bitmap));
    bIsTarget = true;
    strFileName = "DragAndDrop";
    Text = strProgName + " - " + strFileName;
    Invalidate();
}
protected override void OnMouseDown(MouseEventArgs mea)
{
    if (image != null)
    {
        bIsTarget = false;
        DragDropEffects dde = DoDragDrop(image,
            DragDropEffects.Copy | DragDropEffects.Move);
        if (dde == DragDropEffects.Move && !bIsTarget)
            image = null;
    }
}
The OnDragOver and OnDragDrop methods are similar to the DragOver and DragDrop event handlers in NotepadClone. ImageDrop also becomes a drag-and-drop source by calling DoDragDrop during the OnMouseDown method. The program allows Copy and Move actions; if DoDragDrop returns DragDropEffects.Move, the program effectively deletes its copy of the Image object by setting the image variable to null.

An earlier version of this program that I attempted didn't quite work right when I used the program to perform a Move operation on itself. That's because DoDragDrop returns after the OnDragDrop method returns, and the program would delete the image it had just obtained! I defined the bIsTarget variable for this one special case: the program no longer deletes an image moved from itself.
Appendix A: Files and Streams

Overview

Most file I/O support in the .NET Framework is implemented in the System.IO namespace. On first exploration, however—and even during subsequent forays—System.IO can be a forbidding place. It doesn’t help to be reassured that the .NET Framework offers a rich array of file I/O classes and tools. For a C programmer whose main arsenal of file I/O tools consists of library functions such as fopen, fread, fwrite, and fprintf, the .NET file I/O support can seem excessively convoluted and complex.

This appendix is intended to provide a logical progression to guide you through System.IO. I want to identify the really important stuff and also let you know some of the rationale for the multitude of classes.

The .NET Framework distinguishes between files and streams. A file is a collection of data stored on a disk with a name and (often) a directory path. When you open a file for reading or writing, it becomes a stream. A stream is something on which you can perform read and write operations. But streams encompass more than just open disk files. Data coming over a network is a stream, and you can also create a stream in memory. In a console application, keyboard input and text output are also streams.

The Most Essential File I/O Class

If you learn just one class in the System.IO namespace, let it be FileStream. You use this basic class to open, read from, write to, and close files. FileStream inherits from the abstract class Stream, and many of its properties and methods are derived from Stream.

To open an existing file or create a new file, you create an object of type FileStream. These five FileStream constructors have a nice orderly set of overloads:

**FileStream Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileStream(string strFileName, FileMode fm)</td>
</tr>
<tr>
<td>FileStream(string strFileName, FileMode fm, FileAccess fa)</td>
</tr>
<tr>
<td>FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs)</td>
</tr>
<tr>
<td>FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs, int iBufferSize)</td>
</tr>
<tr>
<td>FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs, int iBufferSize, bool bAsync)</td>
</tr>
</tbody>
</table>

There are four additional FileStream constructors based on the operating system file handle. Those are useful for interfacing with existing code. FileMode, FileAccess, and FileShare are all enumerations defined in the System.IO namespace.

The FileMode enumeration indicates whether you want to open an existing file or create a new file and what should happen when the file you want to open doesn’t exist or the file you want to create already exists:

**FileMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateNew</td>
<td>1</td>
<td>Fails if file exists</td>
</tr>
<tr>
<td>Create</td>
<td>2</td>
<td>Deletes file contents if file already exists</td>
</tr>
<tr>
<td>Open</td>
<td>3</td>
<td>Fails if file does not exist</td>
</tr>
<tr>
<td>OpenOrCreate</td>
<td>4</td>
<td>Creates new file if file does not exist</td>
</tr>
</tbody>
</table>
**FileMode Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truncate</td>
<td>5</td>
<td>Fails if file does not exist; deletes contents of file</td>
</tr>
<tr>
<td>Append</td>
<td>6</td>
<td>Fails if file is opened for reading; creates new file if file does not exist; seeks to end of file</td>
</tr>
</tbody>
</table>

By *fail*, I mean that the `FileStream` constructor throws an exception such as `IOException` or `FileNotFoundException`. Almost always, you should call the `FileStream` constructor in a `try` block to gracefully recover from any problems regarding the presumed existence or nonexistence of the file.

Unless you specify a `FileAccess` argument, the file is opened for both reading and writing. The `FileAccess` argument indicates whether you want to read from the file, write to it, or both:

**FileAccess Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1</td>
<td>Fails for <code>FileMode.CreateNew</code>, <code>FileMode.Create</code>, <code>FileMode.Truncate</code>, or <code>FileMode.Append</code></td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
<td>Fails if file is read-only</td>
</tr>
<tr>
<td>ReadWrite</td>
<td>3</td>
<td>Fails for <code>FileMode.Append</code> or if file is read-only</td>
</tr>
</tbody>
</table>

There's one case in which a `FileAccess` argument is required: when you open a file with `FileMode.Append`, the constructor fails if the file is opened for reading. Because files are opened for reading and writing by default, the following constructor always fails:

```csharp
new FileStream(strFileName, FileMode.Append)
```

If you want to use `FileMode.Append`, you also need to include an argument of `FileAccess.Write`:

```csharp
new FileStream(strFileName, FileMode.Append, FileAccess.Write)
```

Unless you specify a `FileShare` argument, the file is open for exclusive use by your process. No other process (or the same process) can open the same file. Moreover, if any other process already has the file open and you don't specify a `FileShare` argument, the `FileStream` constructor will fail. The `FileShare` argument lets you be more specific about file sharing:

**FileShare Enumeration (selection)**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>Allow other processes no access to the file; default</td>
</tr>
<tr>
<td>Read</td>
<td>1</td>
<td>Allow other processes to read the file</td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
<td>Allow other processes to write to the file</td>
</tr>
<tr>
<td>ReadWrite</td>
<td>3</td>
<td>Allow other processes full access to the file</td>
</tr>
</tbody>
</table>

When you only need to read from a file, it's common to allow other processes to read from it also; in other words, `FileAccess.Read` should usually be accompanied by `FileShare.Read`. This courtesy goes both ways: if another process has a file open with `FileAccess.Read` and `FileShare.Read`, your process won't be able to open it unless you specify both flags as well.

**FileStream Properties and Methods**

Once you've opened a file by creating an object of type `FileStream`, you have access to the following five properties implemented in `Stream` that the `FileStream` class overrides:

**Stream Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>CanRead</td>
<td>get</td>
</tr>
</tbody>
</table>
Stream Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>CanWrite</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>CanSeek</td>
<td>get</td>
</tr>
<tr>
<td>long</td>
<td>Length</td>
<td>get</td>
</tr>
<tr>
<td>long</td>
<td>Position</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The first two properties depend on the FileAccess value you used to create the FileStream object. The CanSeek property is always true for open files. The property can return false for other types of streams (such as network streams).

The Length and Position properties are applicable only to seekable streams. Notice that both Length and Position are long integers, and in theory allow file sizes up to $9 \times 10^{12}$, or 9 terabytes, which should be a sufficient maximum file size for at least a couple years.

Setting the Position property is a straightforward way of seeking in the file. (I'll discuss a more conventional Seek method shortly.) For example, if fs is an object of type FileStream, you can seek to the 100th byte in the file with the statement

```csharp
fs.Position = 100;
```

You can seek to the end of a file (for appending to the file) with the statement

```csharp
fs.Position = fs.Length;
```

All the following methods implemented by Stream are overridden by FileStream:

**Stream Methods (selection)**

```csharp
int ReadByte()
int Read(byte[] abyBuffer, int iBufferOffset, int iCount)
void WriteByte(byte byValue)
void Write(byte[] abyBuffer, int iBufferOffset, int iCount)
long Seek(long lOffset, SeekOrigin so)
void SetLength(long lSize);
void Flush()
void Close()
```

You can read either individual bytes with ReadByte or multiple bytes with Read. Both methods return an int value, but that value means different things to each of the methods. ReadByte normally returns the next byte from the file cast to an int without sign extension. For example, the byte 0xFF becomes the integer 0x000000FF, or 255. A return value of -1 indicates an attempt to read past the end of the file.

Read returns the number of bytes read into the buffer, up to iCount. For files, Read returns the same value as the iCount argument unless iCount is greater than the remaining number of bytes in the file. A return value of 0 indicates that there are no more bytes to be read in the file. For other types of streams (network streams, for example), Read can return a value less than iCount but always at least 1 unless the entire stream has been read. The second argument to Read and Write is an offset into the buffer, not an offset into the stream!

The Seek method is similar to the file-seeking functions in C. The SeekOrigin enumeration defines where the lOffset argument to the Seek method is measured from:

**SeekOrigin Enumeration**
If the stream is writable and seekable, the `SetLength` method sets a new length for the file, possibly truncating the contents if the new length is shorter than the existing length. `Flush` causes all data in memory buffers to be written to the file.

Despite what may or may not happen as a result of garbage collection on the `FileStream` object, you should always explicitly call the `Close` method for any files you open.

If you ignore exception handling, in most cases, you can read an entire file into memory—including allocating a memory buffer based on the size of the file—in just four statements:

```csharp
FileStream fs = new FileStream("MyFile", FileMode.Open,
                                    FileAccess.Read, FileShare.Read);
Byte[] abyBuffer = new Byte[fs.Length];
fs.Read(abyBuffer, 0, (int) fs.Length);
fs.Close();
```

I say "in most cases" because this code assumes the file is less than $2^{31}$ bytes (or 2 gigabytes). That assumption comes into play in the casting of the last argument of the `Read` method from a 64-bit `long` to a 32-bit `int`. If the file is larger than 2 gigabytes, you'll have to read it in multiple calls to `Read`. (But you probably shouldn't even be trying to read a multigigabyte file entirely into memory!)

`FileStream` is an excellent choice for a traditional hex-dump program.

```csharp
HexDump.cs
//-------------------------------
// HexDump.cs © 2001 by Charles Petzold
//-------------------------------
using System;
using System.IO;

class HexDump
{
    public static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0)
        {
            Console.WriteLine("Syntax: HexDump file1 file2 ...");
            return 1;
        }
        foreach (string strFileName in astrArgs)
            DumpFile(strFileName);

        return 0;
    }
    protected static void DumpFile(string strFileName)
    {
```
FileStream fs;

try {
    fs = new FileStream(strFileName, FileMode.Open,
                        FileAccess.Read, FileShare.Read);
} catch (Exception exc) {
    Console.WriteLine("HexDump: {0}", exc.Message);
    return;
} 
Console.WriteLine(strFileName);
DumpStream(fs);
fs.Close();
}
protected static void DumpStream(Stream stream) {
    byte[] abyBuffer = new byte[16];
    long lAddress = 0;
    int iCount;

    while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0) {
        Console.WriteLine(ComposeLine(lAddress, abyBuffer, iCount));
        lAddress += 16;
    }
}
public static string ComposeLine(long lAddress, byte[] abyBuffer,
                                  int iCount)
{
    string str = String.Format("{0:X4}-{1:X4}  ",
                                (uint) lAddress / 65536, (ushort) lAddress);

    for (int i = 0; i < 16; i++)
    {
        str += (i < iCount) ?
            String.Format("{0:X2}" , abyBuffer[i]) : " ";

        str += (i == 7 && iCount > 7) ? "-" : " ";
    }
    str += " ";

    for (int i = 0; i < 16; i++)
    {

This program uses the version of `Main` that has a single argument. The argument is an array of strings, each of which is a command-line argument to the program. Unlike the `main` function in C, the `Main` method in C# doesn't include an argument count and also doesn't include the program name among the arguments. If you run the program like so:

```
HexDump file1.cs file2.exe
```

then the argument to `Main` is a string array with two elements. Any wildcards in the arguments are not automatically expanded. (I'll get to wildcard expansion later in this appendix.)

Once HexDump successfully opens each file, the program uses the `Read` method to read 16-byte chunks from the file, and then HexDump's `ComposeLine` method displays them. I've reused the `ComposeLine` method in the HeadDump program in Chapter 16.

`FileStream` has a couple more features I want to mention briefly. For file sharing, you can lock and unlock sections of the file for exclusive use:

```
FileStream Methods (selection)

void Lock(long lPosition, long lLength)
void Unlock(long lPosition, long lLength)
```

If the file system supports asynchronous reading and writing, and if you use the last constructor in the table shown earlier with a last argument of `true`, you can use the `BeginRead`, `EndRead`, `BeginWrite`, and `EndWrite` methods to read from and write to the file asynchronously.

**The Problem with FileStream**

I asserted earlier that `FileStream` is the most essential class in `System.IO` because it opens files and lets you read and write bytes. What could be more basic and vital than that?

The problem, however, is that C# is not nearly as flexible as C in casting. For example, a C programmer might read an `int` from a file by taking the address of an integer variable and casting it to a byte pointer for the `fread` function. But casting something else to a byte array won't work in C#. The `Read` and `Write` methods in `FileStream` work with byte arrays and nothing but byte arrays.

Of course, because the byte is the lowest common denominator, you can always read bytes and assemble them into other basic data types (such as `char` or `int`), and you can disassemble basic types into bytes in preparation for writing. Would you like to do this yourself? I didn't think so.

So, unless reading and writing arrays of bytes is entirely satisfactory to you, you probably can't limit your knowledge of file I/O to the `FileStream` class. As I'll explain shortly, you use the `StreamReader` and `StreamWriter` classes for reading and writing text files, and `BinaryReader` and `BinaryWriter` for reading and writing binary files of types other than byte arrays.

**Other Stream Classes**

The `FileStream` class is one of several classes descended from the abstract class `Stream`. For a class that can't be instantiated, `Stream` plays a very important role in the .NET Framework. This hierarchy diagram shows six classes descended from `Stream`:
The stream classes with an asterisk are defined in namespaces other than `System.IO`.

In addition, a number of methods in other classes scattered throughout the .NET Framework return objects of type `Stream`. For example, as I'll demonstrate later in this appendix, a .NET program that reads files from the Web does so using a `Stream` object. A program in Chapter 11 demonstrates that you can also load image files (such as JPEGs) from streams.

For performance purposes, the `FileStream` class creates a buffered stream. An area of memory is maintained so that every call to `ReadByte`, `Read`, `WriteByte`, and `Write` doesn't necessarily result in a call to the operating system to read from or write to the file.

If you have a `Stream` object that isn't a buffered stream, you can convert it to a buffered stream using the `BufferedStream` class.

The `MemoryStream` class lets you create an expandable area of memory that you can access using the `Stream` methods. I demonstrate how to use the `MemoryStream` class in the `CreateMetafileMemory` program in Chapter 23 and in several programs in Chapter 24.

**Reading and Writing Text**

One important type of file is the text file, which consists entirely of lines of text separated by end-of-line markers. The `System.IO` class has specific classes to read and write text files. Here's the object hierarchy:

Although these classes are not descended from `Stream`, they almost certainly make use of the `Stream` class.

The two classes I'm going to focus on here are `StreamReader` and `StreamWriter`, which are designed for reading and writing text files or text streams. The two other nonabstract classes are `StringReader` and `StringWriter`, which are not strictly file I/O classes. They use similar methods to
read to and write from strings. I discuss these classes briefly at the end of Appendix C and demonstrate the `StringWriter` class in the EnumMetafile program in Chapter 23.

Text may seem to be a very simple form of data storage, but in recent years, text has assumed a layer of complexity as a result of the increased use of Unicode.

The `System.Char` data type in .NET—and the `char` alias in C#—is a 16-bit value representing a character in the Unicode character set. The .NET `System.String` type (and the C# `string` alias) represents a string of Unicode characters. But what happens when you write strings from a C# program to a file? Do you want to write them as Unicode? That makes sense only if every application that reads the file you create expects to be reading Unicode! You probably want to avoid Unicode if you know that other applications reading the file are anticipating encountering 8-bit ASCII characters.

The first 256 characters in Unicode are the same as the 128 characters of ASCII and the 128 characters of the ISO Latin Alphabet No. 1 extension to ASCII. (The combination of these two character sets is often referred to in Windows API documentation as the ANSI character set.) For example, the capital `A` is 0x41 in ASCII and 0x0041 in Unicode. Unicode strings that contain exclusively (or mostly) ASCII contain a lot of zeros. These zeros cause problems for a lot of traditional C-based and UNIX-based programs because those programs interpret a zero byte as a string-termination character.

To alleviate these problems, the `StreamWriter` class lets you have control over how the Unicode strings in your C# program are converted for storage in a file. You assert this control via classes defined in the `System.Text` namespace. Similarly, `StreamReader` lets your program read text files in various formats and convert the text from the files to Unicode strings in your program.

Let's look at `StreamWriter` first. You use this class to write to new or existing text files.

Four of the `StreamWriter` constructors let you create an object of type `StreamWriter` based on a filename:

```csharp
StreamWriter(string strFileName)
StreamWriter(string strFileName, bool bAppend)
StreamWriter(string strFileName, bool bAppend, Encoding enc)
StreamWriter(string strFileName, bool bAppend, Encoding enc, int iBufferSize)
```

These constructors open the file for writing, probably using a `FileStream` constructor internally. By default, if the file exists, the contents will be destroyed. The `bAppend` argument allows you to override that default action. The remaining constructors create an object of type `StreamWriter` based on an existing `Stream` object:

```csharp
StreamWriter(Stream stream)
StreamWriter(Stream stream, Encoding enc)
StreamWriter(Stream stream, Encoding enc, int iBufferSize)
```

If you use a constructor without an `Encoding` argument, the resultant `StreamWriter` object will not store strings to the file in a Unicode format with 2 bytes per character. Nor will it convert your strings to ASCII. Instead, the `StreamWriter` object will store strings in a format known as UTF-8, which is something I'll go over shortly.
If you use one of the `StreamWriter` constructors with an `Encoding` argument, you need an object of type `Encoding`, which is a class defined in the `System.Text` namespace. It's easiest (and in many cases, sufficient) to use one of the static properties of the `Encoding` class to obtain this object:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding</td>
<td>Default</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>Unicode</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>BigEndianUnicode</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>UTF8</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>UTF7</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>ASCII</td>
<td>get</td>
</tr>
</tbody>
</table>

The `Encoding` argument to the `StreamWriter` constructor can also be an instance of one of the classes in `System.Text` that derive from `Encoding`, which are `ASCIIEncoding`, `UnicodeEncoding`, `UTF7Encoding`, and `UTF8Encoding`. The constructors for these classes often have a few options, so you may want to check them out if the static properties aren't doing precisely what you want.

When you specify an encoding of `Encoding.Unicode`, each character is written to the file in 2 bytes with the least significant byte first, in accordance with the so-called little-endian architecture of Intel microprocessors. The file or stream begins with the bytes 0xFF and 0xFE, which correspond to the Unicode character 0xFEFF, which is defined in the Unicode standard as the byte order mark (BOM).

An encoding of `Encoding.BigEndianUnicode` stores the most significant byte of each character first. The file or stream begins with the bytes 0xFE and 0xFF, which also correspond to the Unicode character 0xFEFF. The Unicode character 0xFFFE is intentionally undefined so that applications can determine the byte ordering of a Unicode file from its first two bytes.

If you want to store strings in Unicode but you don't want the byte order marks emitted, you can instead obtain an `Encoding` argument for the `StreamWriter` constructor by creating an object of type `UnicodeEncoding`:

```csharp
new UnicodeEncoding(bBigEndian, bIncludeByteOrderMark)
```

Set the two Boolean arguments appropriately.

UTF-8 is a character encoding designed to represent Unicode characters without using any zero bytes (and hence, to be C and UNIX friendly). UTF stands for `UCS Transformation Format`. UCS stands for `Universal Character Set`, which is another name for ISO 10646, a character-encoding standard with which Unicode is compatible.

In UTF-8, each Unicode character is translated to a sequence of 1 to 6 nonzero bytes. Unicode characters in the ASCII range (0x0000 through 0x007F) are translated directly to single-byte values. Thus, Unicode strings that contain only ASCII are translated to ASCII files. UTF-8 is documented in RFC 2279. (RFC stands for Request for Comments. RFCs are documentations of Internet standards. You can obtain RFCs from many sources, including the Web site of the Internet Engineering Task Force, [http://www.ietf.org](http://www.ietf.org).)

When you specify `Encoding.UTF8`, the `StreamWriter` class converts the Unicode text strings to UTF-8. In addition, it writes the three bytes 0xEF, 0xBB, and 0xBF to the beginning of the file or stream. These bytes are the Unicode BOM converted to UTF-8.

If you want to use UTF-8 encoding but you don't want those three bytes emitted, don't use `Encoding.UTF8`. Use `Encoding.Default` instead or one of the constructors that doesn't have an `Encoding` argument. These options also provide UTF-8 encoding, but the three identification bytes are not emitted.

Alternatively, you can create an object of type `UTF8Encoding` and pass that object as the argument to `StreamWriter`. Use

```csharp
new UTF8Encoding()
```
or

```csharp
new UTF8Encoding(false)
```

to suppress the three bytes, and use

```csharp
new UTF8Encoding(true)
```

to emit the identification bytes.

UTF-7 is documented in RFC 2152. Unicode characters are translated to a sequence of bytes that has an upper bit of 0. UTF-7 is intended for environments in which only 7-bit values can be used, such as e-mail. Use `Encoding.UTF7` in the `StreamWriter` constructor for UTF-7 encoding. No identification bytes are involved with UTF-7.

When you specify an encoding of `Encoding.ASCII`, the resultant file or stream contains only ASCII characters, that is, characters in the range 0x00 through 0x7F. Any Unicode character not in this range is converted to a question mark (ASCII code 0x3F). This is the only encoding in which data is actually lost.

The `StreamWriter` class has a few handy properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>BaseStream</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>Encoding</td>
<td>get</td>
</tr>
<tr>
<td>bool</td>
<td>AutoFlush</td>
<td>get/set</td>
</tr>
<tr>
<td>string</td>
<td>NewLine</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The `BaseStream` property returns either the `Stream` object you used to create the `StreamWriter` object or the `Stream` object that the `StreamWriter` class created based on the filename you supplied. If the base stream supports seeking, you can use that object to perform seeking operations on that stream.

The `Encoding` property returns the encoding you specified in the constructor or `UTF8Encoding` if you specified no encoding. Setting `AutoFlush` to `true` performs a flush of the buffer after every write.

The `NewLine` property is inherited from `TextWriter`. By default, it's the string "\n\n" (carriage return and line feed), but you can change it to "\n" (line feed). If you change it to anything else, the files won't be readable by `StreamReader` objects.

The versatility of the `StreamWriter` class involves the `Write` and `WriteLine` methods that the class inherits from `TextWriter`:

```csharp
TextWriter Methods (selection)
```

```csharp
void Write(...)  
void WriteLine(...)  
void Flush()  
void Close()
```

`TextWriter` supports (and `StreamWriter` inherits) 17 versions of `Write` and 18 versions of `WriteLine` that let you specify any object as an argument to the method. The object you specify is converted to a string by the use of its `ToString` method. The `WriteLine` method follows the string with an end-of-line marker. A version of `WriteLine` with no arguments writes just an end-of-line marker. The `Write` and `WriteLine` methods also include versions with formatting strings, just as the `Console.Write` and `Console.WriteLine` methods do.
Here's a tiny program that appends text to the same file every time you run the program.

```csharp
using System;
using System.IO;

class StreamWriterDemo
{
    public static void Main()
    {
        string strFileName = "StreamWriterDemo.txt";
        bool append = true;

        StreamReader sr = new StreamReader(strFileName, append);
        sr.WriteLine("You ran the StreamWriterDemo program on {0}", DateTime.Now);
        sr.Close();
    }
}
```

Notice the `true` argument to the constructor, indicating that the file will be appended to. The Unicode strings in the program are converted to UTF-8, but they will appear to be ASCII.

I mentioned the `Console` class a moment ago. The input and output devices in that class are defined as objects of type `TextWriter`. Try inserting the following lines at the beginning of the `Main` method in `HexDump`:

```csharp
StreamWriter sw = new StreamWriter("prn", false, Encoding.ASCII);
Console.SetOut(sw);
```

You'll also need to add a `using` statement:

```csharp
using System.Text;
```

Now all the output from the program goes to the printer.

The `StreamReader` class is for reading text files or streams. There are five constructors for opening a text file for reading:

```csharp
StreamReader(string strFileName)
StreamReader(string strFileName, Encoding enc)
StreamReader(string strFileName, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect, int iBufferSize)
```

There is an additional set of five constructors for creating a `StreamReader` object based on an existing stream:
**StreamReader Constructors (selection)**

StreamReader(Stream stream)
StreamReader(Stream stream, Encoding enc)
StreamReader(Stream stream, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect, int iBufferSize)

If you set `bDetect` to `true`, the constructor will attempt to determine the encoding of the file from the first two or three bytes. Or you can specify the encoding explicitly. If you set `bDetect` to `true` and also specify an encoding, the constructor will use the specified encoding only if it can't detect the encoding of the file. (For example, ASCII and UTF-7 can't be differentiated by inspection because they don't begin with a BOM and both contain only bytes in the range 0x00 through 0x7F.)

The `StreamReader` class has the following two, read-only properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>BaseStream</td>
<td>get</td>
</tr>
<tr>
<td>Encoding</td>
<td>CurrentEncoding</td>
<td>get</td>
</tr>
</tbody>
</table>

The `CurrentEncoding` property may change between the time the object is constructed and the first read operation performed on the file or stream because the object has knowledge of identification bytes only after the first read.

Here are the methods to peek, read, and close text files:

**StreamReader Methods (selection)**

```csharp
int Peek()
int Read()
int Read(char[] achBuffer, int iBufferOffset, int iCount)
string ReadLine()
string ReadToEnd()
void Close()
```

The `Peek` and the first `Read` methods both return the next character in the stream or −1 if the end of the stream has been reached. You must explicitly cast the return value to a `char` if the return value is not −1. The second `Read` method returns the number of characters read or 0 if the end of the stream has been reached.

The `ReadLine` method reads the next line up to the next end-of-line marker and strips the end-of-line characters from the resultant string. The method returns a zero-length character string if the line of text contains only an end-of-line marker; the method returns `null` if the end of the stream has been reached.

`ReadToEnd` returns everything from the current position to the end of the file. The method returns `null` if the end of the stream has been reached.

Here's a program that assumes the command-line argument is a URI (Universal Resource Identifier) of an HTML file (or other text file) on the Web. It obtains a `Stream` for that file using some boilerplate code involving the `WebRequest` and `WebResponse` classes. It then constructs a `StreamReader`
object from that stream, uses \texttt{ReadLine} to read each line, and then displays each line using \texttt{Console.WriteLine} with a line number.

\textbf{HtmlDump.cs}

\begin{verbatim}
using System;
using System.IO;
using System.Net;

class HtmlDump
{
    public static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0)
        {
            Console.WriteLine("Syntax: HtmlDump URI");
            return 1;
        }

        WebRequest webreq;
        WebResponse webres;

        try
        {
            webreq = WebRequest.Create(astrArgs[0]);
            webres = webreq.GetResponse();
        }
        catch (Exception exc)
        {
            Console.WriteLine("HtmlDump: {0}", exc.Message);
            return 1;
        }

        if (webres.ContentType.Substring(0, 4) != "text")
        {
            Console.WriteLine("HtmlDump: URI must be a text type.");
            return 1;
        }

        Stream stream = webres.GetResponseStream();
        StreamReader strrdr = new StreamReader(stream);
        string strLine;
        int iLine = 1;

        while ((strLine = strrdr.ReadLine()) != null)
        {
            Console.WriteLine("{1}", iLine, strLine);
            iLine += 1;
        }
    }
}
\end{verbatim}
Binary File I/O

By definition, any file that's not a text file is a binary file. I've already discussed the FileStream class, which lets you read and write bytes. But most binary files consist of data types that are stored as multiple bytes. Unless you want to write code that constructs and deconstructs integers and other types from their constituent bytes, you'll want to take advantage of the BinaryReader and BinaryWriter classes, both of which are derived solely from Object:

The constructors for these classes require a Stream object. If you want to use a file with these classes, create a new FileStream object (or obtain one from some other means) first. For the BinaryWriter class, the Encoding you optionally specify affects the storage of text in the stream:

**BinaryWriter Constructors**

- BinaryWriter(Stream stream)
- BinaryWriter(Stream stream, Encoding enc)

The constructors for BinaryReader are identical:

**BinaryReader Constructors**

- BinaryReader(Stream stream)
- BinaryReader(Stream stream, Encoding enc)

Both classes have a single read-only property named BaseStream that is the Stream object you specified in the constructor.

The Write methods in BinaryWriter are defined for all the basic types as well as for arrays of bytes and characters.

**BinaryWriter Public Methods**

- void Write(...)
- void Write(byte[] abyBuffer, int iBufferOffset, int iBytesToWrite)
- void Write(char[] achBuffer, int iBufferOffset, int iBytesToWrite)
- long Seek(int iOffset, SeekOrigin so)
- void Flush()
void Close()

You can use an object of any basic type (bool, byte, sbyte, byte[], char, char[], string, short, ushort, int, uint, long, ulong, float, double, or decimal) as an argument to Write.

These methods do not store any information about the type of the data. Each type uses as many bytes as necessary. For example, a float is stored in 4 bytes. A bool requires 1 byte. The sizes of arrays are not stored. A 256-element byte array is stored in 256 bytes.

Strings stored in the file are preceded by the byte length stored as a 7-bit encoded integer. (The 7-bit integer encoding uses as many bytes as necessary to store an integer in 7-bit chunks. The first byte of storage is the lowest 7 bits of the integer, and so forth. The high bit of each byte is 1 if there are more bytes. The BinaryWriter class includes a protected method named Write7BitEncodedInt that performs this encoding.)

The Close method closes the underlying stream that the BinaryWriter object is based on.

The BinaryReader class has separate methods to read all the various types.

**BinaryReader Methods (selection)**

```csharp
bool ReadBoolean()
byte ReadByte() 
byte[] ReadBytes(int iCount)
sbyte ReadSByte()
char ReadChar()
char[] ReadChars(int iCount)
short ReadInt16()
int ReadInt32()
long ReadInt64()
ushort ReadUInt16()
uint ReadUInt32()
ulong ReadUInt64()
float ReadSingle()
double ReadDouble()
decimal ReadDecimal()
```

These methods throw an exception of type EndOfStreamException if the end of the stream has been reached. In most cases, your program will know the format of a binary file it's accessing and can avoid end-of-stream conditions. However, for maximum protection, you should put your read statements in try blocks in case you encounter corrupted files.

You can also read individual characters, or arrays of bytes or characters:

**BinaryReader Methods (selection)**

```csharp
int PeekChar()
int Read()
void Read(byte[] abyBuffer, int iBufferOffset, int iBytesToRead)
void Read(char[] achBuffer, int iBufferOffset, int iBytesToRead)
void Close()
```
The `PeekChar` and `Read` methods involve characters, not bytes, and will assume that the file is UTF-8 encoded if you don't explicitly indicate an encoding in the constructor. The methods return −1 if the end of the stream has been reached.

If you have experience with file I/O in C programs, you're probably familiar with common techniques to read and write data structures in a binary format. For example, you may define a structure like so:

```c
typedef struct
{
    int       i;
    float     f;
    char      ch[10];
    int       j;
    float     g;
} STRUCTDEF;
```

and a variable of type `STRUCTDEF` like this:

```
STRUCTDEF mystruct;
```

If you open a file with `fopen` and name the `FILE` pointer `file`, you can then write out the contents of the structure using `fwrite`, as here,

```c
fwrite(&mystruct, sizeof(STRUCTDEF), 1, file);
```

and read it back in similarly:

```c
fread(&mystruct, sizeof(STRUCTDEF), 1, file);
```

This job is so easy because C stores the contents of a structure as just a block of memory. The first argument of the `fwrite` and `fread` functions is defined as a `void` pointer, so you can specify a pointer to anything.

With C#, you don't have quite this much casting freedom. You'll probably want to take a completely different (and more structured) approach to reading and writing binary data. Instead of defining structures such as `STRUCTDEF`, you'll be defining classes. When you save an instance of a class to a file, you want to save sufficient information to re-create that object when you read the file. In a well-designed C# class, you'll probably be saving all the properties of the class that are necessary to re-create the object.

Let's assume you have a class named `SampleClass` that has three properties necessary to re-create the object: a `float` named `Value`, a `string` named `Text`, and an object of type `Fish` stored as a property named `BasicFish`. (`Fish` is another class you've created.) `SampleClass` also has a constructor defined to create a new object from these three items:

```c
public SampleClass(float fValue, string strText, Fish fish)
{
    Value = fValue;
    Text = strText;
    BasicFish = fish;
}
```

Let's also assume that you need to use a binary file to store information that consists of many objects, including objects of type `SampleClass`. Each class you create can implement both an instance method named `Write` and a static method named `Read`. Here's the `Write` method for `SampleClass`. Notice the `BinaryWriter` argument.

```c
public void Write(BinaryWriter bw)
{
    bw.Write(Value);
    bw.Write(Text);
    BasicFish.Write(bw);
}
```
Because the Value and Text properties are basic types, this method can simply call the Write method of BinaryWriter for them. But for the BasicFish property, it must call the similar Write method you’ve also implemented in the Fish class, passing to it the BinaryWriter argument.

The Read method is static because it must create an instance of SampleClass after reading binary data from the file:

```csharp
public static SampleClass Read(BinaryReader br)
{
    float fValue = br.ReadSingle();
    string strText = br.ReadString();
    Fish fish = Fish.Read(br);
    return new SampleClass(fValue, strText, fish);
}
```

Notice that the Fish class must also have a similar static Read method.

**The Environment Class**

Let's leave the System.IO namespace briefly to take a look at the Environment class, which is defined in the System namespace. Environment has a collection of miscellaneous properties and methods that are useful for obtaining information about the machine on which the program is running and the current user logged on to the machine. As its name suggests, the Environment class also allows a program to obtain environment strings. (I make use of this latter facility in the EnvironmentVars program in Chapter 18.)

Two methods in Environment provide information about the file system:

```csharp
Environment Static Methods (selection)

string[] GetLogicalDrives()
string GetFolderPath(Environment.SpecialFolder sf)
```

I have a fairly normal system with a CD-ROM drive and an Iomega Zip drive, so on my machine, GetLogicalDrives returns the following four strings, in this order:

A:
C:
D:
E:

The argument to GetFolderPath is a member of the Environment.SpecialFolder enumeration. The rightmost column in the following table indicates the return string from GetFolderPath on a machine running the default installation of Windows 2000, where I've used an ellipsis to indicate that the return string includes the user's name (which is the same as the value returned from the static property Environment.UserName).

<table>
<thead>
<tr>
<th>Environment.SpecialFolder Enumeration</th>
<th>Value</th>
<th>Common Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>2</td>
<td>C:\Documents and Settings...\Start Menu\Programs</td>
</tr>
<tr>
<td>Personal</td>
<td>5</td>
<td>C:\Documents and Settings...\My Documents</td>
</tr>
<tr>
<td>Favorites</td>
<td>6</td>
<td>C:\Documents and Settings...\Favorites</td>
</tr>
<tr>
<td>Startup</td>
<td>7</td>
<td>C:\Documents and Settings...\Start Menu\Programs\Startup</td>
</tr>
<tr>
<td>Recent</td>
<td>8</td>
<td>C:\Documents and Settings...\Recent</td>
</tr>
</tbody>
</table>
Oddly enough, the SpecialFolder enumeration is defined within the Environment class. Instead of calling GetFolderPath as

```
Environment.GetFolderPath(SpecialFolder.Personal)    // Won't work!
```

you need to preface SpecialFolder with the class in which it's defined:

```
Environment.GetFolderPath(Environment.SpecialFolder.Personal)
```

The Environment class also includes a couple properties that relate to the file system and file I/O:

### Environment Static Properties (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>SystemDirectory</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>CurrentDirectory</td>
<td>get/set</td>
</tr>
</tbody>
</table>

The SystemDirectory property returns the same string as the GetFolderPath method with the Environment.SpecialFolder.System argument.

The CurrentDirectory property lets a program obtain or set the current drive and directory for the application. When setting the directory, you can use a relative directory path, including the ".." string to indicate the parent directory. To change to the root directory of another drive, use the drive letter like so:

```
Environment.CurrentDirectory = "D:\";
```

If the current drive and directory are on a drive other than C and you use

```
Environment.CurrentDirectory = "C:"
```

the current directory is set to the last current directory on drive C before the current drive was changed to something other than C. This technique doesn't seem to work with other drives. The call

```
Environment.CurrentDirectory = "D:"
```

always seems to set the current directory as the root directory of drive D.
As you'll see shortly, other classes defined in the `System.IO` namespace have equivalents to `GetLogicalDrives` and `CurrentDirectory`.

**File and Path Name Parsing**

At times, you need to parse and scan filenames and path names. For example, your program may have a fully qualified filename and you may need just the directory or the drive. The `Path` class, defined in the `System.IO` namespace, consists solely of static methods and static read-only fields that ease jobs like these.

In the following table, the right two columns show sample return values from the methods when the `strFileName` argument is the indicated string at the top of the column. In these examples, I'm assuming the current directory is C:\Docs.

<table>
<thead>
<tr>
<th>Method</th>
<th>\DirA\MyFile</th>
<th>DirA\MyFile.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool IsPathRooted(string strFileName)</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>bool HasExtension(string strFileName)</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>string GetFileName(string strFileName)</td>
<td>MyFile</td>
<td>MyFile.txt</td>
</tr>
<tr>
<td>string GetFileNameWithoutExtension(string strFileName)</td>
<td>MyFile</td>
<td>MyFile</td>
</tr>
<tr>
<td>string GetExtension(string strFileName)</td>
<td>.txt</td>
<td></td>
</tr>
<tr>
<td>string GetDirectoryName(string strFileName)</td>
<td>\DirA</td>
<td>DirA</td>
</tr>
<tr>
<td>string GetFullPath(string strFileName)</td>
<td>C:\DirA\MyFile</td>
<td>C:\Docs\DirA\MyFile.txt</td>
</tr>
<tr>
<td>string GetPathRoot(string strFileName)</td>
<td>\</td>
<td></td>
</tr>
</tbody>
</table>

What's interesting here is that neither DirA nor MyFile has to exist for these methods to work. The methods are basically performing string manipulation, possibly in combination with the current directory.

The following two methods return a new path and filename:

**Path Static Methods (selection)**

```csharp
string Combine(string strLeftPart, string strRightPart)
string ChangeExtension(string strFileName, string strNewExtension)
```

The `Combine` method joins together a path name (on the left) with a path and/or filename (on the right). Use `Combine` rather than string concatenation for this job. Otherwise, you have to worry about whether a backslash is the end of the left part or the beginning of the right part. The `ChangeExtension` method simply changes the filename extension from one string to another. Include a period in the new extension. Set the `strNewExtension` argument to `null` to remove the extension.

The following methods obtain an appropriate directory for storing temporary data and a fully qualified unique filename the program can use to store temporary data:

**Path Static Methods (selection)**

```csharp
```
string GetTempPath()
string GetTempFileName()

If you must do your own file and path name parsing, don't hard-code characters that you think you'll encounter in the strings. Use the following static read-only fields of Path instead:

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Accessibility</th>
<th>Windows Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>PathSeparator</td>
<td>read-only</td>
<td>;</td>
</tr>
<tr>
<td>char</td>
<td>VolumeSeparatorChar</td>
<td>read-only</td>
<td>:</td>
</tr>
<tr>
<td>char</td>
<td>DirectorySeparatorChar</td>
<td>read-only</td>
<td>\</td>
</tr>
<tr>
<td>char</td>
<td>AltDirectorySeparatorChar</td>
<td>read-only</td>
<td>/</td>
</tr>
<tr>
<td>char[]</td>
<td>InvalidPathChars</td>
<td>read-only</td>
<td>&quot;&lt;&gt;</td>
</tr>
</tbody>
</table>

**Parallel Classes**

Another common file I/O job is obtaining a list of all files and subdirectories in a directory. Historically, this job has always been a bit awkward. The standard libraries associated with the C programming language didn't include such a facility, probably because UNIX directory lists were text files that programs could directly access and parse.

Four classes provide you with information about files and directories: Directory, File, DirectoryInfo, and FileInfo. All four of these classes (as well as the Path class I just described) are sealed and can't be inherited. Here's the class hierarchy:

```
Object
  Path
  Directory
  File
  MarshalByRefObject
    FileSystemInfo (abstract)
      DirectoryInfo
      FileInfo
```

Directory and File can't be instantiated; the two classes consist solely of static methods.

DirectoryInfo and FileInfo contain no static methods or properties, and you must obtain an object of type DirectoryInfo or FileInfo to use these classes. Both classes derive from the abstract class FileSystemInfo, so they share some properties and methods.

As the names suggest, Directory and DirectoryInfo provide similar methods, except that the Directory methods are static and require an argument that is a directory name. The DirectoryInfo properties and methods are not static; the constructor argument indicates the directory name to which the properties and methods apply.

Similarly, File and FileInfo provide corresponding methods, except that you indicate a particular filename in the static File method calls and you create an instance of File by specifying a filename in the constructor.

If you need information about a particular file, you may wonder whether it's best to use File or FileInfo (or similarly for directories, whether to use Directory or DirectoryInfo). If you need only one
item of information, it's probably easiest to use the appropriate static method in File or Directory. However, if you need multiple items, it makes more sense to create an object of type FileInfo or DirectoryInfo and then use the instance properties and methods. But don't feel pressured to use one class in preference to the other.

Working with Directories

Let's begin with the Directory and DirectoryInfo classes. The following three static methods of the Directory class have no equivalents in the DirectoryInfo class:

**Directory Static Methods (selection)**

```
string[] GetLogicalDrives()
string GetCurrentDirectory()
void SetCurrentDirectory(string strPath)
```

These methods essentially duplicate the static GetLogicalDrives method and the CurrentDirectory property of the Environment class.

To use any of the properties or methods of the DirectoryInfo class, you need a DirectoryInfo object. One of the ways in which you can obtain such an object is by using the DirectoryInfo constructor:

**DirectoryInfo Constructor**

```
DirectoryInfo(string strPath)
```

The directory doesn't have to exist. Indeed, if you want to create a new directory, creating an object of type DirectoryInfo is a first step.

After creating an object of type DirectoryInfo, you can determine whether the directory exists. Even if the directory doesn't exist, you can obtain certain information about the directory as if it did exist. The two rightmost columns of the following table show examples. The column heading is the string passed to the DirectoryInfo constructor. The current directory is assumed to be C:\Docs.

**DirectoryInfo Properties (selection)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
<th>DirA</th>
<th>DirA\DirB.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Exists</td>
<td>get</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>Name</td>
<td>get</td>
<td>DirA</td>
<td>DirB.txt</td>
</tr>
<tr>
<td>string</td>
<td>FullName</td>
<td>get</td>
<td>C:\Docs\DirA</td>
<td>C:\Docs\DirA\DirB.txt</td>
</tr>
<tr>
<td>string</td>
<td>Extension</td>
<td>get</td>
<td>C:\Docs\DirA\DirB.txt</td>
<td>.txt</td>
</tr>
<tr>
<td>DirectoryInfo</td>
<td>Parent</td>
<td>get</td>
<td>C:\Docs</td>
<td>C:\Docs\DirA</td>
</tr>
<tr>
<td>DirectoryInfo</td>
<td>Root</td>
<td>get</td>
<td>C:\ \</td>
<td>C:\</td>
</tr>
</tbody>
</table>

FullName and Extension are inherited from the FileSystemInfo class.

A few of these properties are also duplicated as static methods in the Directory class. Because they are static methods, they require an argument indicating the path name you're interested in:

**Directory Static Methods (selection)**

```
bool Exists(string strPath)
DirectoryInfo GetParent(string strPath)
```
string GetDirectoryRoot(string strPath)

I mentioned earlier that you can create a DirectoryInfo object based on a directory that doesn't exist. You can then create that directory on the disk by calling the Create method, or you can create a subdirectory of the directory:

**DirectoryInfo Methods (selection)**

void Create()
DirectoryInfo CreateSubdirectory(string strPath)
void Refresh()

Notice that the CreateSubdirectory call returns another DirectoryInfo object with information about the new directory. If the indicated directory already exists, no exception is thrown. The directory used to create the DirectoryInfo object or passed to CreateSubdirectory can contain multiple levels of directory names.

If the directory doesn't exist when you create the DirectoryInfo object and you then call Create, the Exists property won't suddenly become true. You must call the Refresh method (inherited from FileSystemInfo) to refresh the DirectoryInfo information.

The Directory class also has a static method to create a new directory:

**Directory Static Methods (selection)**

DirectoryInfo CreateDirectory(string strPath)

You can delete directories using the Delete method of DirectoryInfo:

**DirectoryInfo Delete Methods**

void Delete()
void Delete(bool bRecursive)

The methods have corresponding static versions in the Directory class:

**Directory Delete Static Methods**

void Delete(string strPath)
void Delete(string strPath, bool bRecursive)

If you use the second version of Delete in either table and you set the bRecursive argument to true, the method also erases all files and subdirectories in the indicated directory. Otherwise, the directory must be empty or an exception will be thrown.

Although the following information is more useful in connection with files, this table of four properties completes our survey of the DirectoryInfo properties:

**DirectoryInfo Properties (selection)**
<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileAttributes</td>
<td>Attributes</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>CreationTime</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>LastAccessTime</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>LastWriteTime</td>
<td>get/set</td>
</tr>
</tbody>
</table>

These properties are all inherited from the FileSystemInfo class, and except for Attributes, they are all duplicated by static methods in the Directory class:

**Directory Static Methods (selection)**

```csharp
DateTime GetCreationTime(string strPath)
DateTime GetLastAccessTime(string strPath)
DateTime GetLastWriteTime(string strPath)
void SetCreationTime(string strPath, DateTime dt)
void SetLastAccessTime(string strPath, DateTime dt)
void SetLastWriteTime(string strPath, DateTime dt)
```

The DateTime structure is defined in the System namespace. FileAttributes is a collection of bit flags defined as an enumeration:

**FileAttributes Enumeration**

<table>
<thead>
<tr>
<th>Member</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadOnly</td>
<td>0x00000001</td>
</tr>
<tr>
<td>Hidden</td>
<td>0x00000002</td>
</tr>
<tr>
<td>System</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Directory</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Archive</td>
<td>0x00000020</td>
</tr>
<tr>
<td>Device</td>
<td>0x00000040</td>
</tr>
<tr>
<td>Normal</td>
<td>0x00000080</td>
</tr>
<tr>
<td>Temporary</td>
<td>0x00000100</td>
</tr>
<tr>
<td>SparseFile</td>
<td>0x00000200</td>
</tr>
<tr>
<td>ReparsePoint</td>
<td>0x00000400</td>
</tr>
<tr>
<td>Compressed</td>
<td>0x00000800</td>
</tr>
<tr>
<td>Offline</td>
<td>0x00001000</td>
</tr>
<tr>
<td>NotContentIndexed</td>
<td>0x00002000</td>
</tr>
<tr>
<td>Encrypted</td>
<td>0x00004000</td>
</tr>
</tbody>
</table>

Directories always have the Directory bit (0x10) set.

To move a directory and all its contents to another location on the same disk, you can use the MoveTo method:

**DirectoryInfo Methods (selection)**

```csharp
DirectoryInfo.MoveTo(string newPath)
```
void MoveTo(string strPathDestination)

Or you can use the static Move method in the Directory class:

**Directory Static Methods (selection)**

```csharp
void Move(string strPathSource, string strPathDestination)
```

With either method call, the destination must not currently exist.

The remaining methods of DirectoryInfo and Directory obtain an array of all the files and subdirectories in a directory, or only those directories and files that match a specified pattern using wildcards (question marks and asterisks). Here are the six methods of DirectoryInfo:

**DirectoryInfo Methods (selection)**

```csharp
DirectoryInfo[] GetDirectories()
DirectoryInfo[] GetDirectories(string strPattern)
FileInfo[] GetFiles()
FileInfo[] GetFiles(string strPattern)
FileSystemInfo[] GetFileSystemInfos()
FileSystemInfo[] GetFileSystemInfos(string strPattern)
```


You'll recall that FileSystemInfo is the parent class for both DirectoryInfo and FileInfo.

The Directory class has a similar set of six methods, but these all return arrays of strings:

**Directory Static Methods (selections)**

```csharp
string[] GetDirectories(string strPath)
string[] GetDirectories(string strPath, string strPattern)
string[] GetFiles(string strPath)
string[] GetFiles(string strPath, string strPattern)
string[] GetFileSystemEntries(string strPath)
string[] GetFileSystemEntries(string strPath, string strPattern)
```

We're now fully equipped to enhance the HexDump program shown earlier so that it works with wildcard file specifications on the command line. Here's WildCardHexDump.

**WildCardHexDump.cs**

```csharp
//---------------------------------------------------------------
// WildCardHexDump.cs © 2001 by Charles Petzold
//---------------------------------------------------------------
using System;
using System.IO;
```
class WildCardHexDump: HexDump
{
    public new static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0)
        {
            Console.WriteLine("Syntax: WildCardHexDump file1 file2 ...");
            return 1;
        }
        foreach (string str in astrArgs)
            ExpandWildCard(str);
        return 0;
    }
    static void ExpandWildCard(string strWildCard)
    {
        string[] astrFiles;
        try
        {
            astrFiles = Directory.GetFiles(strWildCard);
        }
        catch
        {
            try
            {
                string strDir  = Path.GetDirectoryName(strWildCard);
                string strFile = Path.GetFileName(strWildCard);
                if (strDir == null || strDir.Length == 0)
                    strDir = ".";
                astrFiles = Directory.GetFiles(strDir, strFile);
            }
            catch
            {
                Console.WriteLine(strWildCard + ": No Files found!");
                return;
            }
        }
        if (astrFiles.Length == 0)
            Console.WriteLine(strWildCard + ": No files found!");
        foreach (string strFile in astrFiles)
Besides normal wildcards, I wanted to be able to specify just a directory name as an argument. For example, I wanted

```
WildCardHexDump c:\
```

to be the equivalent of

```
WildCardHexDump c:\*.*
```

The `ExpandWildCard` method begins by attempting to obtain all the files in the particular command-line argument:

```
astrFiles = Directory.GetFiles(strWildCard);
```

This call will work if `strWildCard` specifies only a directory (such as "c:\"). Otherwise, it throws an exception. That's why it's in a `try` block. The `catch` block assumes that the command-line argument has path and filename components, and it obtains these components using the static `GetDirectoryName` and `GetFileName` methods of `Path`. However, the `GetFiles` method of `Directory` doesn't want a first argument that is `null` or an empty string. Before calling `GetFiles`, the program avoids that problem by setting the path name to ".", which indicates the current directory.

### File Manipulation and Information

Like the `Directory` and `DirectoryInfo` classes, the `File` and `FileInfo` classes are very similar and share many properties and methods. Like the `Directory` class, all the methods in the `File` class are static, and the first argument to every method is a string that indicates the path name of the file. The `FileInfo` class inherits from `FileSystemInfo`. You create an object of type `FileInfo` based on a filename that could include a full or a relative directory path.

#### FileInfo Constructor

```
FileInfo(string strFileName)
```

The file doesn't have to exist. You can determine whether the file exists, and also some information about it, with the following read-only properties:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Exists</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>Name</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>FullName</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>Extension</td>
<td>get</td>
</tr>
<tr>
<td>string</td>
<td>DirectoryName</td>
<td>get</td>
</tr>
<tr>
<td>DirectoryInfo</td>
<td>Directory</td>
<td>get</td>
</tr>
<tr>
<td>long</td>
<td>Length</td>
<td>get</td>
</tr>
</tbody>
</table>

Only one of these properties is duplicated in the `File` class:

#### File Methods

```
bool Exists(string strFileName)
```
FileInfo has four additional properties that reveal the attributes of the file and the dates the file was created, last accessed, and last written to:

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileAttributes</td>
<td>Attributes</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>CreationTime</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>LastAccessTime</td>
<td>get/set</td>
</tr>
<tr>
<td>DateTime</td>
<td>LastWriteTime</td>
<td>get/set</td>
</tr>
</tbody>
</table>

These properties, all of which are inherited from FileSystemInfo, are all duplicated by static methods in the File class:

**File Static Methods (selection)**

- FileAttributes GetAttributes(string strFileName)
- DateTime GetCreationTime(string strFileName)
- DateTime GetLastAccessTime(string strFileName)
- DateTime GetLastWriteTime(string strFileName)
- void SetAttributes(string strFileName, FileAttributes fa)
- void SetCreationTime(string strFileName, DateTime dt)
- void SetLastAccessTime(string strFileName, DateTime dt)
- void SetLastWriteTime(string strFileName, DateTime dt)

The following methods let you copy, move, or delete the file. I've included the Refresh method here, which refreshes the object's properties after you've made a change to the file:

**FileInfo Methods (selection)**

- FileInfo CopyTo(string strFileName)
- FileInfo CopyTo(string strFileName, bool bOverwrite)
- void MoveTo(string strFileName)
- void Delete()
- void Refresh()

The copy, move, and delete facilities are duplicated in the File class:

**File Static Methods (selection)**

- void Copy(string strFileNameSrc, string strFileNameDst)
- void Copy(string strFileNameSrc, string strFileNameDst, bool bOverwrite)
- void Move(string strFileNameSrc, string strFileNameDst, bool bOverwrite)
- void Delete(string strFileName)

And finally, the File and FileInfo classes have several methods to open files:

**FileInfo Methods (selection)**
FileStream Create()
FileStream Open(FileMode fm)
FileStream Open(FileMode fm, FileAccess fa)
FileStream Open(FileMode fm, FileAccess fa, FileShare fs)
FileStream OpenRead()
FileStream OpenWrite()
StreamReader OpenText()
StreamWriter CreateText()
StreamWriter AppendText()

These are handy if you've just obtained an array of FileInfo objects from a GetFiles call on a DirectoryInfo object and you want to poke your nose into each and every file.

You can also use the corresponding static methods implemented in the File class:

**File Static Methods (selection)**

FileStream Create(string strFileName)
FileStream Open(string strFileName, FileMode fm)
FileStream Open(string strFileName, FileMode fm, FileAccess fa)
FileStream Open(string strFileName, FileMode fm, FileAccess fa, FileShare fs)
FileStream OpenRead(string strFileName)
FileStream OpenWrite(string strFileName)
StreamReader OpenText(string strFileName)
StreamWriter CreateText(string strFileName)
StreamWriter AppendText(string strFileName)

However, these methods don't provide any real advantage over using the appropriate constructors of the FileStream, StreamReader, or StreamWriter class. Indeed, their very presence in the File class was initially one of the aspects of the entire System.IO namespace that I found most confusing. It doesn't make sense to use a class like File merely to obtain an object of type FileStream so that you can then use FileStream properties and methods. It's easier to use just a single class if that's sufficient for your purposes.
Appendix B: Math Class

Working with numbers is the most fundamental programming task. The Microsoft .NET Framework and C# add a few features to numbers that may be new to veteran C programmers. In this appendix, I'll discuss those features as well as the all-important Math class, which contains methods that are equivalents of functions declared in the C Math.h header file.

Numeric Types

The C# language supports 11 numeric types that fall into three categories: integral, floating point, and decimal:

<table>
<thead>
<tr>
<th>C# Numeric Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>128</td>
</tr>
</tbody>
</table>

In a C# program, an integer literal (that is, a number written without a decimal point) is assumed to be an int unless its value is larger than a maximum int, in which case the value of the number is used to determine its type. The number is assumed to be a uint, long, or ulong (in that order) depending on its value. A literal with a decimal point (or that includes an exponent indicated with an E or e followed by a number) is assumed to be a double. You can use the following suffixes on numeric literals to clarify your intentions.

<table>
<thead>
<tr>
<th>Suffixes for Numeric Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>uint</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>ulong</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>decimal</td>
</tr>
</tbody>
</table>

The C# type names are aliases for structures defined in the System class of the .NET Framework. These structures are all derived from ValueType, which itself derives from Object.

<table>
<thead>
<tr>
<th>.NET Numeric Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
</tr>
<tr>
<td>Bits</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>128</td>
</tr>
</tbody>
</table>
The `SByte`, `UInt16`, `UInt32`, and `UInt64` types are not compliant with the Common Language Specification (CLS). What that means is that a programming language can be compliant with the CLS without supporting these types. If you write code that you want to be usable by all CLS-compliant languages (such as in DLLs), do not use signed bytes or unsigned 16-bit, 32-bit, or 64-bit integers.

### Checking Integer Overflow

Consider the following code:
```
short s = 32767;
s += 1;
```

Here's another one:
```
ushort us = 0;
us -= 1;
```

These are examples of integer overflow and underflow, and both these snippets of code are perfectly legal in C as well as C# (by default anyway).

In the first case, a signed integer is being incremented past its maximum value. Due to the manner in which integers are stored in memory, the result will be \(-32768\). In the second case, an unsigned integer is being decremented below zero, and the result is 65535.

Sometimes programmers take advantage of integer overflow and underflow, and sometimes programmers fall victim to overflow and underflow bugs. To separate clever techniques from nasty bugs, C# allows you to optionally check for integer overflow and underflow.

To subject an entire program to runtime checking of overflow and underflow, use the following compiler switch:
```
/checked+
```

The following compiler switch results in the default option:
```
/checked-
```

In Visual C# .NET, you can set this compiler switch by first invoking the Property Pages dialog box for the project. On the left side of the dialog box, select Build from Configuration Properties. On the right side of the dialog box, set the option Check For Arithmetic Overflow/Underflow to True.

When you enable runtime checking of overflow and underflow, the increment and decrement operations just shown will raise an exception of type `OverflowException`.

Within your C# program, you can override the compiler setting by using the keywords `checked` and `unchecked`. You follow the keyword with an expression enclosed in parentheses, or a statement or group of statements in curly brackets. For example, the code
```
short s = 32767;
checked
{
    s += 1;
}
```

will raise an exception regardless of the compiler switch. You'll probably want to enclose `checked` blocks within `try` blocks.

So far, I've been speaking solely of runtime checking of integer overflow and underflow. By default, the compiler will flag compile-time overflow and underflow as an error regardless of the compiler switch you use. For example, the statement
```
short s = 32767 + 1;
```
is always a compile-time error because the addition is evaluated during compilation. However, it is possible to use the unchecked keyword to override compile-time overflow and underflow checking. For example, suppose you define two const integers like so:

```csharp
const int i1 = 65536;
const int i2 = 65536;
```

The expression

```csharp
int i3 = i1 * i2;
```

will normally cause a compile-time error. Because `i1` and `i2` are both const values, the compiler attempts to evaluate the expression and encounters an overflow. A compiler switch won't override that behavior, but the unchecked keyword will:

```csharp
int i3 = unchecked (i1 * i2);
```

That statement will compile fine and execute without raising an exception.

You should probably write your program to compile and run correctly under either compiler option. Whenever there's a danger of overflow or underflow that you want to catch, enclose the statement in a checked block within a try block. Whenever you don't care about overflow or underflow, or you want to exploit overflow or underflow in some way, enclose the statement in an unchecked block.

Regardless of any compiler switches or the presence of the checked and unchecked keywords, integer division by zero always raises a DivideByZeroException.

**The Decimal Type**

The C# numeric type that is entirely new to C programmers is the decimal type, which uses 16 bytes (128 bits) to store each value. The 128 bits break down into a 96-bit integral part, a 1-bit sign, and a scaling factor that can range from 0 through 28. Mathematically, the scaling factor is a negative power of 10 and indicates the number of decimal places in the number.

Don't confuse the decimal type with a binary-coded decimal (BCD) type. In a BCD type, each decimal digit is stored using 4 bits. The decimal type stores the number as a binary integer.

For example, if you define a decimal equal to 12.34, the number is stored as the integer 0x4D2 (or 1234) with a scaling factor of 2. A BCD encoding would store the number as 0x1234.

As long as a decimal number has 28 significant digits (or fewer) and 28 decimal places (or fewer), the decimal data type stores the number exactly. This is not true with floating point! If you define a float equal to 12.34, it’s essentially stored as the value 0xC570A4 (or 12,939,428) divided by 0x100000 (or 1,048,576), which is only approximately 12.34. Even if you define a double equal to 12.34, it’s stored as the value 0x18AE147AE147AE (or 6,946,802,425,218,990) divided by 0x20000000000000 (or 562,949,953,421,312), which again only approximately equals 12.34.

And that's why you should use decimal when you're performing calculations where you don't want pennies to mysteriously crop up and disappear. The floating-point data type is great for scientific and engineering applications but often undesirable for financial ones.

If you want to explore the internals of the decimal, you can make use of the following constructor:

**Decimal Constructors (selection)**

```csharp
Decimal(int iLow, int iMiddle, int iHigh, bool bNegative, byte byScale)
```

Although defined as integers, the first three arguments of the constructor are treated as unsigned integers to form a composite 96-bit unsigned integer. The byScale argument (which can range from 0 through 28) is the number of decimal places. For example, the expression

```csharp
new Decimal(123456789, 0, 0, false, 5)
```

creates the decimal number
The largest positive decimal number is
new Decimal(-1, -1, -1, false, 0)

or
79,228,162,514,264,337,593,543,950,335

which you can also obtain from the MaxValue field of the Decimal structure:
Decimal.MaxValue

The smallest decimal number closest to 0 is
new Decimal(1, 0, 0, false, 28)

which equals
0.0000000000000000000000000001

or
1 × 10⁻²⁸

If you divide this number by 2 in a C# program, the result is 0.

It's also possible to obtain the bits used to store a decimal value:

Decimal Static Methods (selection)

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[] GetBits(decimal mValue)</td>
</tr>
</tbody>
</table>

This method returns an array of four integers. The first, second, and third elements of the array are the low, medium, and high components of the 96-bit unsigned integer. The fourth element contains the sign and the scaling factor: bits 0 through 15 are 0; bits 16 through 23 contain a scaling value between 0 and 28; bits 24 through 30 are 0; and bit 31 is 0 for positive and 1 for negative.

If you have a decimal number named mValue, you can execute the statement
int[] ai = Decimal.GetBits(mValue);

If ai[3] is negative, the decimal number is negative. The scaling factor is
(ai[3] >> 16) & 0xFFFF

I already indicated how floating-point representation is often only approximate. When you start performing arithmetic operations on floating-point numbers, the approximations can get worse. Almost everyone who has used floating point is well aware that a number that should be 4.55 (for example) is often stored as 4.549999 or 4.550001.

The decimal representation is much better behaved. For example, suppose m1 is defined like so:

decimal m1 = 12.34;

Internally, m1 has an integer part of 1234 and a scaling factor of 2. Also, suppose m2 is defined like this:

decimal m2 = 56.789;

The integer part is 56789, and the scaling factor is 3. Now add these two numbers:

decimal m3 = m1 + m2;

Internally, the integer part of m1 is multiplied by 10 (to get 12340), and the scaling factor is set to 3. Now the integer parts can be added directly: 12340 plus 56789 equals 69129 with a scaling factor of 3. The actual number is 69.129. Everything is exact.
Now multiply the two numbers:

decimal m4 = m1 * m2;

Internally, the two integral parts are multiplied (1234 times 56789 equals 70,077,626), and the scaling factors are added (2 plus 3 equals 5). The actual numeric result is 700.77626. Again, the calculation is exact.

When dividing...well, division is messy no matter how you do it. But for the most part, when using decimal, you have much better control over the precision and accuracy of your results.

**Floating-Point Infinity and NaNs**

The two floating-point data types—float and double—are defined in accordance with the ANSI/IEEE Std 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*.

A float value consists of a 24-bit signed mantissa and an 8-bit signed exponent. The precision is approximately seven decimal digits. Values range from

\[-3.402823 \times 10^{38}\]

to

\[3.402823 \times 10^{38}\]

The smallest possible float value greater than 0 is

\[1.401298 \times 10^{-45}\]

You can obtain these three values as fields in the Single structure:

<table>
<thead>
<tr>
<th>Single Structure Constant Fields (selection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>float</td>
</tr>
</tbody>
</table>

A double value consists of a 53-bit signed mantissa and an 11-bit signed exponent. The precision is approximately 15 to 16 decimal digits. Values range from

\[-1.79769313486232 \times 10^{308}\]

to

\[1.79769313486232 \times 10^{308}\]

The smallest possible double value greater than 0 is

\[4.94065645841247 \times 10^{-324}\]

The MinValue, MaxValue, and Epsilon fields are also defined in the Double structure.

Here's some code that divides a floating-point number by 0:

```csharp
float f1 = 1;
float f2 = 0;
float f3 = f1 / f2;
```

If these were integers, a DivideByZeroException would be raised. But these are IEEE floating-point numbers. An exception is *not* raised. Indeed, floating-point operations never raise exceptions. Instead, in this case, f3 takes on a special value. If you use Console.WriteLine to display f3, it will display the word

**Infinity**

If you change the initialization of f1 to −1, Console.WriteLine will display
In the IEEE standard, positive infinity and negative infinity are legitimate values of floating-point numbers. You can even perform arithmetic on infinite values. For example, the expression $1 / f3$
equals 0.

If you change the initialization of $f1$ in the preceding code to 0, then $f3$ will equal a value known as Not a Number, which is universally abbreviated as NaN and pronounced "nan." Here's how `Console.WriteLine` displays a NaN:

NaN

You can also create a NaN by adding a positive infinity to a negative infinity or by a number of other calculations.

Both the `Single` and `Double` structures have static methods to determine whether a `float` or `double` value is infinity or NaN. Here are the methods in the `Single` structure:

```
Single Structure Static Methods (selection)

bool IsInfinity(float fValue)
bool IsPositiveInfinity(float fValue)
bool IsNegativeInfinity(float fValue)
bool IsNaN(float fValue)
```

For example, the expression

```csharp
Single.IsInfinity(fVal)
```

returns `true` if `/Val` is either positive infinity or negative infinity.

The `Single` structure also has constant fields that represent these values:

```
Single Structure Constant Fields (selection)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>PositiveInfinity</td>
</tr>
<tr>
<td>float</td>
<td>NegativeInfinity</td>
</tr>
<tr>
<td>float</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

Identical fields are defined in the `Double` structure. These values correspond to specific bit patterns defined in the IEEE standard.

The **Math Class**

The `Math` class in the `System` namespace consists solely of a collection of static methods and the following two constant fields:

```
Math Constant Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>PI</td>
<td>3.14159265358979</td>
</tr>
<tr>
<td>double</td>
<td>E</td>
<td>2.71828182845905</td>
</tr>
</tbody>
</table>
```

`Math.PI`, of course, is the ratio of the circumference of a circle to its diameter, and `Math.E` is the limit of
as $n$ approaches infinity.

Most of the methods in the `Math` class are defined only for `double` values. However, some methods are defined for integer and `decimal` values as well. The following two methods are defined for every numeric type:

**Math Static Methods (selection)**

- `type Max(numeric-type n1, numeric-type n2)`
- `type Min(numeric-type n1, numeric-type n2)`

The two values must be the same type.

The following two methods are defined for `float`, `double`, `decimal`, and all signed integer types:

**Math Static Methods (selection)**

- `int Sign(signed-type s)`
- `type Abs(signed-type s)`

The `Sign` method returns 1 if the argument is positive, −1 if the argument is negative, and 0 if the argument is 0. The `Abs` method returns the argument if it's 0 or positive, and the negative value of the argument if the argument is negative.

The `Abs` method is the only method of the `Math` class that can throw an exception, and then only for integral arguments, and only for one particular value for each integral type. If the argument is the `MinValue` of the particular integral type (for example, −32768 for `short`), then an `OverflowException` is raised because 32768 can't be represented by a `short`.

The following methods perform various types of rounding on `double` and `decimal` values:

**Math Static Methods (selection)**

- `double Floor(double dValue)`
- `double Ceiling(double dValue)`
- `double Round(double dValue)`
- `double Round(double dValue, int iDecimals)`
- `decimal Round(decimal mValue)`
- `decimal Round(decimal mValue, int iDecimals)`

*Floor* returns the largest whole number less than or equal to the argument; *Ceiling* returns the smallest whole number greater than or equal to the argument. The call

```
Math.Floor(3.5)
```

returns 3, and

```
Math.Ceiling(3.5)
```
returns 4. The same rules apply to negative numbers. The call
Math.Floor(-3.5)
returns −4, and
Math.Ceiling(-3.5)
returns −3.

The *Floor* method returns the nearest whole number in the direction of negative infinity, and that's why it's sometimes also known as *rounding toward negative infinity*; likewise, *Ceiling* returns the nearest whole number in the direction of positive infinity and is sometimes called *rounding toward positive infinity*. It's also possible to round toward 0, which is to obtain the nearest whole number closest to 0. You round toward 0 by casting to an integer. The expression
(int) 3.5 returns 3, and
(int) −3.5 returns −3. Rounding toward 0 is sometimes called *truncation*.

The *Round* methods with a single argument return the whole number nearest to the argument. If the argument to *Round* is midway between two whole numbers, the return value is the nearest even number. For example, the call
Math.Round(4.5)
returns 4, and
Math.Round(5.5)
returns 6. You can optionally supply an integer that indicates the number of decimal places in the return value. For example,
Math.Round(5.285, 2)
returns 5.28.

**Floating-Point Remainders**

Much confusion surrounds functions that calculate floating-point remainders. The C# remainder or modulus operator (%) is defined for all numeric types. (In C, the modulus operator is not defined for *float* and *double*, the *fmod* function must be used instead.) Here's a C# statement using *float* numbers with the remainder operator:

```csharp
fResult = fDividend % fDivisor;
```

The sign of *fResult* is the same as the sign of *fDividend*, and *fResult* can be calculated with the formula

```csharp
fResult = fDividend - n * fDivisor
```

where *n* is the largest possible integer less than or equal to *fDividend/ fDivisor*. For example, the expression

```csharp
4.5 % 1.25
```

equals 0.75. Let's run through the calculation. The expression 4.5 / 1.25 equals 3.6, so *n* equals 3.

The quantity 4.5 minus (3 times 1.25) equals 0.75.

The IEEE standard defines a remainder a little differently, where *n* is the integer closest to *fDividend/ fDivisor*. You can calculate a remainder in accordance with the IEEE standard using this method:

```csharp
Math静tic Methods (selection)
```

double IEEERemainder(double dDividend, double dDivisor)
The expression
Math.IEEERemainder(4.5, 1.25)
returns \(-0.5\). That's because \(4.5 / 1.25\) equals 3.6, and the closest integer to 3.6 is 4. When \(n\) equals 4, the quantity 4.5 minus (4 times 1.25) equals \(-0.5\).

**Powers and Logarithms**

Three methods of the `Math` class involve powers:

**Math Static Methods (selection)**

- `double Pow(double dBase, double dPower)`
- `double Exp(double dPower)`
- `double Sqrt(double dValue)`

*Pow* calculates the value \(dBase^{dPower}\).

The expression
Math.Exp(dPower)
is equivalent to
Math.Pow(Math.E, dPower)
and the square root function
Math.Sqrt(dValue)
is equivalent to
Math.Pow(dValue, 0.5)

The *Sqrt* method returns NaN if the argument is negative.

The `Math` class has three methods that calculate logarithms:

**Math Static Methods (selection)**

- `double Log10(double dValue)`
- `double Log(double dValue)`
- `double Log(double dValue, double dBase)`

The expression
Math.Log10(dValue)
is equivalent to
Math.Log(dValue, 10)
and
Math.Log(dValue)
is equivalent to
Math.Log(dValue, Math.E)
The logarithm methods return *PositiveInfinity* for an argument of 0 and *NaN* for an argument less than 0.

**Trigonometric Functions**

Trigonometric functions describe the relationship between the sides and angles of triangles. The trig functions are defined for right triangles:

For angle \( \alpha \) in a right triangle where \( x \) is the adjacent leg, \( y \) is the opposite leg, and \( r \) is the hypotenuse, the three basic trigonometric functions are

\[
\sin(\alpha) = \frac{y}{r} \\
\cos(\alpha) = \frac{x}{r} \\
\tan(\alpha) = \frac{y}{x}
\]

Trigonometric functions can also be used to define circles and ellipses. For constant \( r \) and \( \alpha \) ranging from 0 degrees to 360 degrees, the set of coordinates \((x, y)\) where

\[
x = r \cdot \sin(\alpha) \\
y = r \cdot \cos(\alpha)
\]

define a circle centered at the origin with radius \( r \). Chapter 5 shows how to use trigonometric functions to draw circles and ellipses. Trig functions also show up in various graphics exercises in Chapters 13, 15, 17, and 19.

The trigonometric functions in the *Math* class require angles specified in radians rather than degrees. There are \( 2\pi \) radians in 360 degrees. The rationale for using radians can be illustrated by considering the following arc / subtended by angle \( \alpha \):
What is the length of arc \( l \)? Because the circumference of the entire circle equals \( 2\pi r \), the length of arc \( l \) equals \( \frac{\alpha}{360} \cdot 2\pi r \), where \( \alpha \) is measured in degrees. However, if \( \alpha \) is measured in radians, then the length of arc \( l \) simply equals \( \alpha r \). For a unit circle (radius equal to 1), the length of arc \( l \) equals the angle \( \alpha \) in radians. And that’s how the radian is defined: in a unit circle, an arc of length \( l \) is subtended by an angle in radians equal to \( l \).

For example, an angle of 90 degrees in a unit circle subtends an arc with length \( \pi/2 \). Thus, 90 degrees is equivalent to \( \pi/2 \) radians. An angle of 180 degrees is equivalent to \( \pi \) radians. There are \( 2\pi \) radians in 360 degrees.

Here are the three basic trigonometric functions defined in the `Math` class:

### Math Static Methods (selection)

- `double Sin(double dAngle)`
- `double Cos(double dAngle)`
- `double Tan(double dAngle)`

If you have an angle in degrees, multiply by \( \pi \) and divide by 180 to convert to radians:

\[
\text{Math.Sin(Math.PI * dAngleInDegrees / 180)}
\]

The \( \text{Sin} \) and \( \text{Cos} \) methods return values ranging from \(-1\) to \(1\). In theory, the \( \text{Tan} \) method should return infinity at \( \pi/2 \) (90 degrees) and \( 3\pi/2 \) (270 degrees), but it returns very large values instead.

The following methods are inverses of the trigonometric functions. They return angles in radians:

### Math Static Methods (selection)

<table>
<thead>
<tr>
<th>Method</th>
<th>Argument</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double Asin(double dValue)</code></td>
<td>(-1) through (1)</td>
<td>(-\pi/2) through (\pi/2)</td>
</tr>
<tr>
<td><code>double Acos(double dValue)</code></td>
<td>(-1) through (1)</td>
<td>(\pi) through (0)</td>
</tr>
<tr>
<td><code>double Atan(double dValue)</code></td>
<td>(-\infty) through (\infty)</td>
<td>(-\pi/2) through (\pi/2)</td>
</tr>
<tr>
<td><code>double Atan2(double y, double x)</code></td>
<td>(-\infty) through (\infty)</td>
<td>(-\pi) through (\pi)</td>
</tr>
</tbody>
</table>

To convert the return value to degrees, multiply by 180 and divide by \(\pi\).

The \( \text{Asin} \) and \( \text{Acos} \) methods return \(\text{NaN}\) if the argument is not in the proper range. The \( \text{Atan2} \) method uses the signs of the two arguments to determine the quadrant of the angle:

### Atan2 Return Values

<table>
<thead>
<tr>
<th>( y ) Argument</th>
<th>( x ) Argument</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>Positive</td>
<td>0 through (\pi/2)</td>
</tr>
<tr>
<td>Positive</td>
<td>Negative</td>
<td>(\pi/2) through (\pi)</td>
</tr>
<tr>
<td>Negative</td>
<td>Negative</td>
<td>(\pi) through (3\pi/2)</td>
</tr>
<tr>
<td>Negative</td>
<td>Positive</td>
<td>(3\pi/2) through (2\pi)</td>
</tr>
</tbody>
</table>

Less commonly used are the hyperbolic trigonometric functions. While the common trigonometric functions define circles and ellipses, the hyperbolic trig functions define hyperbolas:

### Math Static Methods (selection)

- `double Sinh(double dAngle)`
double Cosh(double dAngle)
double Tanh(double dAngle)

The angle is expressed in hyperbolic radians.
Appendix C: String Theory

Overview

Just about every programming language ever invented implements text strings a little differently. Unlike floating-point numbers, strings are not blessed (or cursed) with an industry standard. The C programming language doesn't even have a separate data type for strings. A string is simply an array of characters terminated with a zero byte. A program references the string by a pointer to the first character in the array. C programmers appreciate the ease with which strings can be manipulated in memory. C programmers are also quite familiar with the ease in which seemingly innocent string manipulations can become nasty bugs.

In C#, the text string is its own data type named `string`, which is an alias for the class `System.String`. The `string` data type is related to the `char` data type, of course: a `string` object can be constructed from an array of characters and also converted into an array of characters. But a `string` and a `char` array are two distinct data types.

C# strings are not zero-terminated. A string has a specific length, and once a string is created, its length can't be changed. Nor can any of the individual characters that make up a string be changed. A C# string is thus said to be immutable. Whenever you need to change a string in some way, you must create another string. Many methods of the `String` class create new strings based on existing strings. Many methods and properties throughout the .NET Framework create and return strings.

Here's a common pitfall: you might expect that there's a method of `String` named `ToUpper` that converts all the characters in a string to uppercase, and that's precisely the case. But for a `string` instance named `str`, you can't just call the method like so:

```csharp
str.ToUpper();   // Won't do anything!
```

Syntactically, this statement is valid, but it has no effect on the `str` variable. Strings are immutable, and hence the characters of `str` can't be altered. The `ToUpper` method creates a new string. You need to assign the return value of `ToUpper` to another string variable:

```csharp
string strUpper = str.ToUpper();
```

Or you could assign it to the same string variable:

```csharp
str = str.ToUpper();
```

In the second case, the original string (the one containing lowercase letters) still exists, but since it's probably no longer referenced anywhere in the program, it becomes eligible for garbage collection.

Here's another example. Suppose you define a string like so:

```csharp
string str = "abcdifg";
```

You can access a particular character of the string by indexing the string variable:

```csharp
char ch = str[4];
```

In this case, `ch` is the character 'i'. But you can't set a particular character of a string:

```csharp
str[4] = 'e';   // Won't work!
```

The indexer property of the `String` class is read-only.

So, how do you replace characters in a C# string? There are a couple ways. The method call

```csharp
str = str.Replace('i', 'e');
```

will replace all occurrences of 'i' with 'e'. Alternatively, you can first call `Remove` to create a new string with one or more characters removed at a specified index with a specified length. For example, the call

```csharp
str = str.Remove(4, 1);
```

removes one character at the fourth position (the 'i'). You can then call `Insert` to insert a new string, which in this case is a single character:
str = str.Insert(4, "e");

Or you can do both jobs in one statement:
str = str.Remove(4, 1).Insert(4, "e");

Despite the use of a single string variable named `str`, the two method calls in this last statement create two additional strings, and the quoted 'e' is yet another string.

Another approach is also possible. You can convert the string into a character array, set the appropriate element of the array, and then construct a new string based on the character array:

```csharp
char[] ach = str.ToCharArray();
ach[4] = 'e';
str = new String(ach);
```

Or you can patch together a new string from substrings:
`str = str.Substring(0, 4) + "e" + str.Substring(5);`

I'll discuss all these `String` class methods more formally in the course of this appendix.

**The char Type**

Each element of a string is a `char`, which is an alias for the .NET structure `System.Char`. A program can specify a single literal character using single quotation marks:
`char ch = 'A';`

Although `Char` is derived from `ValueType`, a `char` variable isn't directly usable as a number. To convert a `char` variable named `ch` to an integer, for example, requires casting:
`int i = (int) ch;`

Character variables have numeric values from 0x0000 through 0xFFFF and refer to characters in the Unicode character set. The book *The Unicode Standard Version 3.0* (Addison-Wesley, 2000) is the essential reference to Unicode.

As in C, the backslash (\) is a special escape character. The character following the backslash has a special interpretation, as shown in the following table:

### C# Control Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\0</td>
<td>Null</td>
<td>0x0000</td>
</tr>
<tr>
<td>\a</td>
<td>Alert</td>
<td>0x0007</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>0x0008</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
<td>0x0009</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td>0x000A</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab</td>
<td>0x000B</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
<td>0x000C</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
<td>0x000D</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
<td>0x0022</td>
</tr>
<tr>
<td>\</td>
<td>Single quote</td>
<td>0x0027</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
<td>0x005C</td>
</tr>
</tbody>
</table>

In addition, you can specify a single Unicode character using the preface `\x` or `\u` followed by a four-digit hexadecimal number. The characters "\x03A9" and "\u03A9" both refer to the Greek capital omega (Ω).
In C, as you know, you can use functions defined in the `ctype.h` header file to determine whether a particular character is a letter, number, control character, or whatever. In C#, you use static methods defined in the `Char` structure. The argument is either a character or a string with an index value.

**Char Static Methods (selection)**

```csharp
bool IsControl(char ch)
bool IsControl(string str, int iIndex)
bool IsSeparator(char ch)
bool IsSeparator(string str, int iIndex)
bool IsWhiteSpace(char ch)
bool IsWhiteSpace(string str, int iIndex)
bool IsPunctuation(char ch)
bool IsPunctuation(string str, int iIndex)
bool IsSymbol(char ch)
bool IsSymbol(string str, int iIndex)
bool IsDigit(char ch)
bool IsDigit(string str, int iIndex)
bool IsNumber(char ch)
bool IsNumber(string str, int iIndex)
bool IsLetter(char ch)
bool IsLetter(string str, int iIndex)
bool IsUpper(char ch)
bool IsUpper(string str, int iIndex)
bool IsLower(char ch)
bool IsLower(string str, int iIndex)
bool IsLetterOrDigit(char ch)
bool IsLetterOrDigit(string str, int iIndex)
bool IsSurrogate(char ch)
bool IsSurrogate(string str, int iIndex)
```

The call

```
Char.IsControl(str[iIndex])
```

is equivalent to

```
Char.IsControl(str, iIndex)
```

You might be able to avoid using these methods for ASCII characters (character values 0x0000 through 0x007F), but these methods also apply to all Unicode characters. The `IsSurrogate` method refers to the area of Unicode with values 0xD800 through 0xDFFF that is reserved for expansion.

The `Char` structure also defines a couple other handy methods. One returns a member of the `UnicodeCategory` enumeration (defined in `System.Globalization`), and the other returns the numeric value of the character converted to a `double`.

**Char Static Methods (selection)**

```csharp
UnicodeCategory GetUnicodeCategory(char ch)
UnicodeCategory GetUnicodeCategory(string str, int iIndex)
double GetNumericValue(char ch)
```
double GetNumericValue(string str, int iIndex)

String Constructors and Properties

In many cases, you'll define a string variable using a literal:

```csharp
string str = "Hello, world!";
```

or a literal inserted right in a function call:

```csharp
Console.WriteLine("Hello, world!");
```

or as the return value from one of the many methods that return string variables. One ubiquitous string-returning method is named `ToString` and converts an object to a string. For example, the expression

```csharp
55.ToString();
```

returns the string "55".

If you preface a string literal with the @ sign, the backslash is not interpreted as an escape character. This technique is handy for specifying directories:

```csharp
string str = @"c:\temp\my file";
```

To include a double quotation mark in such a string, use two double quotation marks in succession.

One of the less common methods of creating a string is by using one of the eight `String` constructors. Five of the `String` constructors involve pointers and are not compliant with the Common Language Specification (CLS). The remaining three `String` constructors create a `String` object by repeating a single character or converting from an array of characters:

**String Constructors (selection)**

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String(char ch, int iCount)</code></td>
</tr>
<tr>
<td><code>String(char[] ach)</code></td>
</tr>
<tr>
<td><code>String(char[] ach, int iStartIndex, int iCount)</code></td>
</tr>
</tbody>
</table>

In the third constructor, `iStartIndex` is an index into the character array and `iCount` indicates a number of characters beginning at that index. The length of the resultant string will equal `iCount`.

The `String` class has just two properties, both of which are read-only:

**String Properties**

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>Length</code></td>
<td><code>get</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td><code>[]</code></td>
<td><code>get</code></td>
</tr>
</tbody>
</table>

The first indicates the number of characters in the string; the second is an indexer that lets you access the individual characters of the string.

You can define a string variable without initializing it:

```csharp
string str1;
```

Any attempt to use that string will cause the compiler to report that the string variable is unassigned. Because `String` is a reference type, you can assign a string variable the value `null`:

```csharp
string str2 = null;
```

What the `null` value means is that no memory has been allocated for the string. Having a `null` value is different from having an empty string:

```csharp
string str3 = "";
```
An empty string has memory allocated for the instance of the string, but the `str3.Length` property equals 0. Attempting to determine the length of a `null` string—making reference to `str2.Length`, for example—causes an exception to be thrown.

You can also initialize a string variable to an empty string using the only public field of the `String` class:

<table>
<thead>
<tr>
<th>Static Field</th>
<th>Type</th>
<th>Field</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td><code>Empty</code></td>
<td></td>
<td>read-only</td>
</tr>
</tbody>
</table>

For example,

```csharp
string str = string.Empty;
```

You can define an array of strings like so:

```csharp
string[] astr = new string[5];
```

An array of five strings is created, each of which is `null`. You can also create an array of initialized strings:

```csharp
string[] astr = { "abc", "defghi", "jkl" };
```

This statement creates an array with three elements; that is, `astr.Length` returns 3. Each string has a specific length; for example, `astr[1].Length` returns 6.

The `String` class implements the `IComparable`, `ICloneable`, `IConvertible`, and `IEnumerable` interfaces, which implies that the `String` class contains certain methods defined in these interfaces. Because `String` implements the `IEnumerable` interface, you can use `String` with the `foreach` statement to enumerate the characters in a string. The statement

```csharp
foreach (char ch in str)
{
    :
}
```

is equivalent to (and quite a bit shorter than)

```csharp
for (int i = 0; i < str.Length; i++)
{
    char ch = str[i];
    :
}
```

In the `foreach` block, `ch` is read-only. In the `for` block, `ch` is not read-only but (as usual) the string characters can't be altered.

After `IEnumerable`, perhaps the next most important interface that `String` implements is `IComparable`, which means that the `String` class implements a method named `CompareTo` that lets you use arrays of strings with the `BinarySearch` and `Sort` methods defined in the `Array` class. I'll go over these methods later in this appendix.

### Copying Strings

There are several ways to copy a string. Perhaps the simplest is using the equals sign:

```csharp
string strCopy = str;
```

Like every class in the .NET Framework, the `String` class inherits the `ToString` method from `Object`. Because the `String` class implements `ICloneable`, it also implements the `Clone` method. These methods provide additional (if somewhat redundant) methods to copy strings:

**String Methods (selection)**
string ToString()
object Clone()

If you use `Clone`, you must cast the result to a `string`:

```csharp
string strCopy = (string) str.Clone();
```

The `String` class also implements a static method that copies a string:

**String Copy Static Method**

```csharp
string Copy(string str)
```

Because `string` is an alias for `System.String`, you can preface the method name with the lowercase `string`:

```csharp
string strCopy = string.Copy(str);
```

or with the fully qualified class name:

```csharp
string strCopy = System.String.Copy(str);
```

If you have a `using System` statement in the program, you can prefix the method name with the uppercase `String` class name:

```csharp
string strCopy = String.Copy(str);
```

Two of the `String` constructors convert a character array to a string. You can also convert a string back to a character array:

**String Methods (selection)**

```csharp
char[] ToCharArray()
char[] ToCharArray(int iStartIndex, int iCount)
void CopyTo(int iStartIndexSrc, char[] achDst, int iStartIndexDst, int iCount)
```

The `ToCharArray` methods create the character array. The `iStartIndex` argument refers to a starting index in the string. To use the `CopyTo` method, the `achDst` array must already exist. The first argument is a starting index for the string; the third argument is a starting index in the character array. The `CopyTo` method is the equivalent of

```csharp
for (int i = 0; i < iCount; i++)
    achDst[iStartIndexDst + i] = str[iStartIndexSrc + i];
```

The `Substring` methods create a new string that is a section of an existing string:

**String Substring Method**

```csharp
string Substring(int iStartIndex)
string Substring(int iStartIndex, int iCount)
```

The first version returns a substring that begins at the index and continues to the end of the string.
Converting Strings

Two methods, each with two versions, convert strings to lowercase or uppercase:

**String Methods (selection)**

- `string ToUpper()`
- `string ToUpper(CultureInfo ci)`
- `stringToLower()`
- `string ToLower(CultureInfo ci)`

The `CultureInfo` class is defined in `System.Globalization` and in this case refers to a particular language as used in a particular country.

Concatenating Strings

It's often necessary to tack together two or more strings, a process known as *string concatenation*. In C, you use the library functions `strcat` and `strncat`. For convenience, the C# addition operator is overloaded to perform string concatenation:

```csharp
string str = str1 + str2;
```

The concatenation operator is convenient for defining a string literal that's a little too long to fit on a single line:

```csharp
string str = "Those who profess to favor freedom and yet depreciate " + 
             "agitation. . .want crops without plowing up the ground, they 
             " + 
             "want rain without thunder and lightning. They want the ocean 
             " + 
             "without the awful roar of its many waters. \x2014 Frederick 
             " + 
             "Douglass";
```

You can also use the `+=` operator to append a string to the end of an existing string:

```csharp
str += "\r\n";
```

The `String` class also defines a static `Concat` method:

**String Concat Static Method (selection)**

- `string Concat(string str1, string str2)`
- `string Concat(string str1, string str2, string str3)`
- `string Concat(string str1, string str2, string str3, string str4)`
- `string Concat(params string[] astr)`

Notice the `params` keyword in the last version of `Concat`. What that keyword means in this case is that you can specify either an array of strings or any number of strings. For example, if you have an array of strings defined as

```csharp
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

and you pass that array to the `Concat` method

```csharp
string str = string.Concat(astr);
```
the result is the string "abcdefgijklmnopqr". Alternatively, you can pass the individual strings directly to the `Concat` method:

```csharp
string str = string.Concat("abc", "def", "ghi", "jkl", "mno", "pqr");
```

Although the `String` class defines `Concat` versions with two, three, four, or a variable number of arguments, only the version with the `params` argument is necessary. That method actually encompasses the other three methods.

Another set of `Concat` methods are the same except with `object` arguments:

**String Concat Static Method (selection)**

```csharp
string Concat(object obj)
string Concat(object obj1, object obj2)
string Concat(object obj1, object obj2, object obj3)
string Concat(params object[] aobj)
```

The `object` arguments are converted to strings by the objects' `ToString` methods. The call

```csharp
string.Concat(55, "-", 33, "=", 55 - 33)
```

creates the string "55-33=22".

It's sometimes necessary to concatenate an array of strings but with some kind of separator between each array element. You can do that using the `Join` static method:

**String Join Static Method**

```csharp
string Join(string strSeparator, string[] astr)
string Join(string strSeparator, string[] astr, int iStartIndex, int iCount)
```

For example, if you have an array of strings defined as

```csharp
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

you might want to create a composite string with end-of-line indicators between each pair. Call

```csharp
string str = string.Join("\r\n", astr);
```

The result is the string

```
abc\r\ndef\r\nghi\r\njkl\r\nmno\r\npqr
```

The separator is not appended following the last string.

The second version of `Join` lets you select a contiguous subset of strings from the array before joining them.

### Comparing Strings

`String` is a class (not a structure), and `string` is a reference type (not a value type). Normally that would imply that the comparison operators (`==` and `!=`) wouldn't work correctly for strings. You'd be comparing object references rather than characters. However, the `==` and `!=` operators have been redefined for strings and work as you'd expect. The expressions

```csharp
(str == "New York")
```

and

```csharp
(str != "New Jersey")
```
return bool values based on a case-sensitive character-by-character comparison.

There are several methods defined in the String class that return bool values indicating the result of a case-sensitive string comparison:

### String Methods (selection)

- bool Equals(string str)
- bool Equals(object obj)
- bool StartsWith(string str)
- bool EndsWith(string str)

If a string is defined as

```csharp
string str = "The end of time";
```

then

```csharp
str.StartsWith("The")
```

returns true but

```csharp
str.StartsWith("the")
```

returns false.

There's also a static version of the Equals method:

### String Static Methods (selection)

- bool Equals(string str1, string str2)

For example, instead of

```csharp
if (str == "New York")
```

you can use

```csharp
if (Equals(str, "New York"))
```

Methods like this one are provided primarily for languages that don't have operators for comparison.

The remaining comparison methods implemented in String, which I'll discuss momentarily, return an integer value that indicates whether one string is less than, equal to, or greater than another string:

### String Comparison Method Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>str1 &lt; str2</td>
</tr>
<tr>
<td>Zero</td>
<td>str1 == str2</td>
</tr>
<tr>
<td>Positive</td>
<td>str1 &gt; str2</td>
</tr>
</tbody>
</table>

Watch out: the comparison methods are defined as returning negative, zero, or positive integers, not −1, 0, or 1.

Usually if you're interested in whether one string is less than or greater than another, it's because you're sorting the strings in some way. And that implies that you probably don't want to perform a comparison based on the strict numeric values of the character codes. For example, you probably want the characters e and é to be regarded as less than F, despite the higher values of their
character codes. Such a comparison is known as a *lexical* comparison rather than a *numeric* comparison.

Here's the relationship among a few select characters when compared numerically:

\[ D < E < F < d < e < f < Ê < É < Ë < ë \]

And here's the lexical comparison:

\[ d < D < e < E < é < É < ë < Ë \]

Is a lexical comparison also case insensitive? Mostly it is. For example, the string "New Jersey" is considered less than "new York" despite the lowercase 'n' in the second string. But when two strings are identical except for case, lowercase letters are considered less than uppercase letters, that is, "the" is less than "The". However, "Them" is less than "then".

In other words, by default, a lexical comparison is case sensitive only when a method must decide whether or not to return 0. Otherwise, it's case insensitive.

The lexical comparison also implies a certain relationship among letters, numbers, and other characters. In general, control characters are considered to be less than single quotes and dashes, which are less than white-space characters. Next comes punctuation and other symbols, digits (0 through 9), and finally letters. A null string is less than the empty string, which is less than any other character. For example,

"New" < "New York" < "Newark"

The nonstatic method `CompareTo` performs a lexical comparison between a string instance and an argument:

```csharp
// String CompareTo Method
int CompareTo(string str2)
int CompareTo(object obj2)
```

The first string is the string object you're calling `CompareTo` on, for example,

`str1.CompareTo(str2)`

The `CompareTo` method with the object argument is necessary to implement the `IComparable` interface. The `CompareTo` method is used by the static `Array.BinarySearch` and `Array.Sort` methods, as I'll discuss shortly.

All the other comparison methods are static. The `CompareOrdinal` methods perform a strict numeric comparison based on the character value:

```csharp
// String CompareOrdinal Static Method
int CompareOrdinal(string str1, string str2)
int CompareOrdinal(string str1, int iStartIndex1, string str2, int iStartIndex2, int iCount)
```

The static `Compare` methods perform a lexical comparison:

```csharp
// String Compare Static Method
int Compare(string str1, string str2)
```
int Compare(string str1, string str2, bool bIgnoreCase)
int Compare(string str1, string str2, bool bIgnoreCase, CultureInfo ci)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase, CultureInfo ci)

The `bIgnoreCase` argument affects the return value only when the two strings are the same except for case. Case-insensitive comparisons are much more useful for searching rather than sorting. The method calls

```csharp
string.Compare("ë", "È")

and

string.Compare("ë", "È", false)
```

both return negative values, but

```csharp
string.Compare("ë", "È", true)
```

returns 0. The calls

```csharp
string.Compare("e", "ë", bIgnoreCase)

and

string.Compare("e", "È", bIgnoreCase)
```

always return negative values, regardless of the presence or value of the `bIgnoreCase` argument.

There is no comparison method implemented in the `String` class that reports that "André" equals "Andre".

**Searching the String**

The C library functions `strchr` and `strstr` search a string for the first occurrence of a specific character or another string and return a pointer to that occurrence. The C# equivalents—which are all versions of the `IndexOf` method—return an index in the source string rather than a pointer.

**String IndexOf Methods**

```csharp
int IndexOf(char ch)
int IndexOf(char ch, int iStartIndex)
int IndexOf(char ch, int iStartIndex, int iCount)
int IndexOf(string str)
int IndexOf(string str, int iStartIndex)
int IndexOf(string str, int iStartIndex, int iCount)
```

You can search for a specific character or another string. The search is case sensitive. The method returns −1 if the character or string isn't found. You can optionally include a starting index and a character count. The return value is measured from the beginning of the string, not from the starting index.

With a string defined as
string str = "hello world";
then
str.IndexOf('o')
returns 4, and
str.IndexOf("wo")
returns 6.

You can also perform the search starting at the end of the string:

**String LastIndexOf Methods**

```csharp
int LastIndexOf(char ch)
int LastIndexOf(char ch, int iStartIndex)
int LastIndexOf(char ch, int iStartIndex, int iCount)
int LastIndexOf(string str)
int LastIndexOf(string str, int iStartIndex)
int LastIndexOf(string str, int iStartIndex, int iCount)
```

Although the methods search from the end of the string, the returned index is still measured from the beginning of the string. For the string shown above, the call
str.LastIndexOf('o')
returns 7, and
str.LastIndexOf("wo")
returns 6.

The following methods have a first argument that is an array of characters. The methods determine the first or last index in the string of a character that matches any character in the array:

**String Methods (selection)**

```csharp
int IndexOfAny(char[] ach)
int IndexOfAny(char[] ach, int iStartIndex)
int IndexOfAny(char[] ach, int iStartIndex, int iCount)
int LastIndexOfAny(char[] ach)
int LastIndexOfAny(char[] ach, int iStartIndex)
int LastIndexOfAny(char[] ach, int iStartIndex, int iCount)
```

If a character array and a string are defined like so:
char[] achVowel = { 'a', 'e', 'i', 'o', 'u' };
string str = "hello world";
then
str.IndexOfAny(achVowel)
returns 1, and
str.LastIndexOfAny(achVowel)
returns 7.

**Trimming and Padding**

Sometimes when processing text files (such as program source code files), it's convenient to remove *white space*, which is the nonvisible characters that separate other elements in the string. The *String* class has methods to do so. For purposes of these methods, white-space characters are assumed to be the following Unicode characters:

<table>
<thead>
<tr>
<th>Unicode White-Space Characters</th>
<th>\textbf{Value}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{0x0009} (tab)</td>
<td>\textbf{0x2003} (em space)</td>
</tr>
<tr>
<td>\textbf{0x000A} (line feed)</td>
<td>\textbf{0x2004} (three-per-em space)</td>
</tr>
<tr>
<td>\textbf{0x000B} (vertical tab)</td>
<td>\textbf{0x2005} (four-per-em space)</td>
</tr>
<tr>
<td>\textbf{0x000C} (form feed)</td>
<td>\textbf{0x2006} (six-per-em space)</td>
</tr>
<tr>
<td>\textbf{0x000D} (carriage return)</td>
<td>\textbf{0x2007} (figure space)</td>
</tr>
<tr>
<td>\textbf{0x0020} (space)</td>
<td>\textbf{0x2008} (punctuation space)</td>
</tr>
<tr>
<td>\textbf{0x00A0} (no-break space)</td>
<td>\textbf{0x2009} (thin space)</td>
</tr>
<tr>
<td>\textbf{0x2000} (en quad)</td>
<td>\textbf{0x200A} (hair space)</td>
</tr>
<tr>
<td>\textbf{0x2001} (em quad)</td>
<td>\textbf{0x200B} (zero-width space)</td>
</tr>
<tr>
<td>\textbf{0x2002} (en space)</td>
<td>\textbf{0x3000} (ideographic space)</td>
</tr>
</tbody>
</table>

You can either use the predefined white-space characters or define your own array of characters.

**String Methods (selection)**

```csharp
string Trim()
string Trim(params char[] ach)
string TrimStart(params char[] ach)
string TrimEnd(params char[] ach)
```

To remove the predefined white-space characters from the beginning and end of a string named `str`, use

```
str.Trim()
```

or

```
str.Trim(null)
```

You can also remove the predefined white-space characters from the beginning of a string, as here:

```
str.TrimStart(null)
```

or the end, as here:

```
str.TrimEnd(null)
```

Alternatively, you can specify the characters (not necessarily white-space characters) you want removed from the beginning or end of a string. You can either define a character array and pass that to the *Trim* (or *TrimStart* or *TrimEnd*) method

```
char[] achTrim = { ',', '-', '_' };
str.Trim(achTrim)
```

or list the characters explicitly in the method call:

```
str.Trim( ',', '-', '_' );
```
Both method calls cause these three characters to be stripped from the beginning and end of the string.

You can also add spaces (or any other character) to the beginning or end of a string to achieve a specified total width:

**String Methods (selection)**

```csharp
string PadLeft(int iTotalLength)
string PadLeft(int iTotalLength, char ch)
string PadRight(int iTotalLength)
string PadRight(int iTotalLength, char ch)
```

**String Manipulation**

Here are some miscellaneous methods that let you insert one string into another, remove a range of characters, and replace a particular character or string within a string. I showed examples of all these methods at the beginning of this appendix:

**String Methods (selection)**

```csharp
string Insert(int iIndex, string strInsert)
string Remove(int iIndex, int iCount)
string Replace(char chOld, char chNew)
string Replace(string strOld, string strNew)
```

You may have had occasion to use the C library function `strtok`. This function is intended to break a string down into *tokens*, which are substrings delimited by certain fixed characters, usually white-space characters. In C, you call `strtok` repeatedly until the source string has no more tokens. In C#, you can do the work of `strtok` with a single call to the `Split` method:

**String Split Method**

```csharp
string[] Split(params char[] achSeparators)
string[] Split(params char[] achSeparators, int iReturnCount)
```

If you set the first argument to `null`, the method uses the set of white-space characters shown earlier.

**Formatting Strings**

As you know from Chapter 1, the first argument of the `Console.Write` or `Console.WriteLine` method can be a string that describes the formatting of the remaining arguments. If these two methods are the C# equivalent of the C `printf` function, the static `Format` method of `String` is the C# equivalent of the C `sprintf` function:

**String Format Static Method (selection)**

```csharp
string Format(string strFormat, object obj0)
string Format(string strFormat, object obj0, object obj1)
string Format(string strFormat, object obj0, object obj1, object obj2)
string Format(string strFormat, params object[] aobj)
```
For example, the following call to `Format`,

```csharp
string str = String.Format("The sum of {0} and {1} is {2}", 2, 3, 2 + 3);
```

creates the string "The sum of 2 and 3 is 5".

**Array Sorting and Searching**

The `String` class implements the `IComparable` interface, which merely requires that it implement the following method:

```csharp
public int CompareTo(object obj)
```

This method is called by two useful static methods of `Array` named `Sort` and `BinarySearch`. You can use these two methods with arrays of objects of any class that implements `IComparable`.

Here are the two basic `Sort` methods:

**Array Sort Static Methods (selection)**

```csharp
void Sort(Array arr)
void Sort(Array arr, int iStartIndex, int iCount)
```

The second version allows you to use a subset of the array. Suppose you define an array of strings like so:

```csharp
string[] astr = { "New Jersey", "New York", "new Mexico", "New Hampshire" };
```

Notice the lowercase `n` in the third string. After calling

```csharp
Array.Sort(astr);
```

the elements of the array are reordered to be "New Hampshire", "New Jersey", "new Mexico", and "New York". Because the `Sort` method uses the `CompareTo` method of `String`, the sorting is case insensitive. However, if the array also included "New Mexico" (with an uppercase `N`), "New Mexico" would be appear after "new Mexico" in the sorted array.

The next two versions of the `Sort` method require two corresponding arrays of equal size, optionally with a starting index and an element count:

**Array Sort Static Methods (selection)**

```csharp
void Sort(Array arrKeys, Array arrItems)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, int iCount)
```

The method sorts the first array and reorders the second array accordingly. I use this version of the `Sort` method in the `SysInfoReflectionStrings` program in Chapter 4 to sort an array of `SystemInformation` property names stored in `astrLabels`:

```csharp
Array.Sort(astrLabels, astrValues);
```

The corresponding array of `SystemInformation` values stored in `astrValues` is also reordered so that the array elements still correspond to each other.
If you want to perform a sort using a method other than `CompareTo`, you use one of the following `Sort` methods:

**Array Sort Static Methods (selection)**

```csharp
void Sort(Array arr, IComparer comp)
void Sort(Array arr, int iStartIndex, int iCount, IComparer comp)
void Sort(Array arrKeys, Array arrItems, IComparer comp)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, iCount, IComparer comp)
```

The argument of type `IComparer` can be an instance of any class that implements the `IComparer` interface. That's not the `String` class! `String` implements the `IComparable` interface, not `IComparer`.

The `IComparer` interface is defined in the `System.Collections` namespace. A class that implements `IComparer` must define the following method:

**IComparer Method**

```csharp
int Compare(object obj1, object obj2)
```

This method is not static, and hence, is not defined in the `String` class. (The only methods named `Compare` implemented in `String` are static methods.)

The `System.Collections` namespace contains two classes that implement `IComparer`, which are `Comparer` (to perform a case-sensitive comparison just like the default) and `CaseInsensitiveComparer` (for a case-insensitive string comparison). Both these classes have a static member named `Default` that returns an instance of the class.

For example, to perform a case-sensitive sort of the string array `astr`, call

```csharp
Array.Sort(astr);
```

or

```csharp
Array.Sort(astr, Comparer.Default);
```

To perform a case-insensitive sort, call

```csharp
Array.Sort(astr, CaseInsensitiveComparer.Default);
```

The case-insensitive compare is much more useful in the `BinarySearch` method rather than the `Sort` method (or when sorting in preparation for a binary search):

**Array BinarySearch Static Method**

```csharp
int BinarySearch(Array arr, object obj)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj)
int BinarySearch(Array arr, object obj, IComparer comp)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj, IComparer comp)
```

To perform a binary search, the array must be sorted. The sorted array of four state names contains the elements
The call
Array.BinarySearch(astr, "New York")
returns 3 because the string is identical to astr[3]. The call
Array.BinarySearch(astr, "New Mexico")
returns −4. The negative number indicates that the string isn't in the array. (Remember, by default
the search is case sensitive!) The complement of the return value is 3, which means that astr[3] is
the next highest element of the array.

The call
Array.BinarySearch(astr, "new Mexico");
returns 2 because the argument matches astr[2]. The call
Array.BinarySearch(astr, "New Mexico", CaseInsensitiveComparer.Default));
performs a case-insensitive search and also returns 2.

The StringBuilder Class

You may wonder if there's a performance penalty associated with frequent re-creations of String
objects. Sometimes there is. Consider the following program, which uses the += operator in 10,000
string-appending operations to construct a large string.

StringAppend.cs

```csharp
using System;

class StringAppend
{
    const int iIterations = 10000;

    public static void Main()
    {
        DateTime dt = DateTime.Now;
        string str = String.Empty;

        for (int i = 0; i < iIterations; i++)
            str += "abcdefghijklmnopqrstuvwxyz\r\n";

        Console.WriteLine(DateTime.Now - dt);
    }
}
```

The program calls the Now method of the DateTime class at the beginning and end to calculate an
elapsed time, which is displayed in hours, minutes, seconds, and units of 100 nanoseconds. (See
Chapter 10 for information about DateTime and related classes.) Each string-appending operation
causes a new String object to be created, which requires another memory allocation. Each previous
string is marked for garbage collection.
How fast this program runs depends on how fast your machine is. It could take about a minute or so.

A better solution in a case like this is the appropriately named `StringBuilder` class, defined in the `System.Text` namespace. Unlike the string maintained by the `String` class, the string maintained by `StringBuilder` can be altered. `StringBuilder` dynamically reallocates the memory used for the string. Whenever the size of the string is about to exceed the size of the memory buffer, the buffer is doubled in size. To convert a `StringBuilder` object to a `String` object, call the `ToString` method.

Here's a revised version of the program, which uses `StringBuilder`.

**StringBuilderAppend.cs**
```csharp
//--------------------------------------------------
// StringBuilderAppend.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Text;

class StringBuilderAppend
{
    const int iIterations = 10000;

    public static void Main()
    {
        DateTime dt = DateTime.Now;
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < iIterations; i++)
        {
            sb.Append("abcdefghijklmnopqrstuvwxyz\r\n");
        }

        string str = sb.ToString();

        Console.WriteLine(DateTime.Now - dt);
    }
}
```

You'll probably find that this program does its work in well under a second. It seems to run in under 1/1000 the time of the original version.

Another efficient approach is to use the `StringWriter` class defined in the `System.IO` namespace. As I mentioned in Appendix A, both `StringWriter` and `StreamWriter` (which you use for writing to text files) derive from the abstract `TextWriter` class. Like `StringBuilder`, `StringWriter` assembles a composite string. The big advantage with `StringWriter` is that you can use the whole array of `Write` and `WriteLine` methods defined in the `TextWriter` class. Here's a sample program that performs the same task as the previous two programs but using a `StringWriter` object.

**StringWriterAppend.cs**
```csharp
//--------------------------------------------------
// StringWriterAppend.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.IO;

class StringWriterAppend
{
    const int iIterations = 10000;

    public static void Main()
    {
        DateTime dt = DateTime.Now;
        StringWriter sb = new StringWriter();

        for (int i = 0; i < iIterations; i++)
        {
            sb.Append("abcdefghijklmnopqrstuvwxyz\r\n");
        }

        string str = sb.ToString();

        Console.WriteLine(DateTime.Now - dt);
    }
}
```
const int iIterations = 10000;

public static void Main()
{
    DateTime dt = DateTime.Now;
    StringWriter sw = new StringWriter();

    for (int i = 0; i < iIterations; i++)
        sw.WriteLine("abcdefghijklmnopqrstuvwxyz");

    string str = sw.ToString();

    Console.WriteLine(DateTime.Now - dt);
}

The speed of this program is comparable to StringBuilderAppend.

There's a lesson in all this. As operating systems, programming languages, class libraries, and frameworks provide an ever increasingly higher level of abstraction, we programmers can sometimes lose sight of all the mechanisms going on beneath the surface. What looks like a simple addition in code can actually involve many layers of low-level activity.

We may be insulated from this low-level activity, but we must train ourselves to still feel the heat. If a particular operation seems slow to you, or to require too much memory, or to involve inordinately convoluted code, try to determine why and then search for an alternative. It's likely that someone has already provided exactly what you need.