

THE EXPERT'S VOICE® IN .NET



Pro .NET 2.0 Graphics Programming

Building Custom Controls using GDI+

Eric White

Apress®

 PreviousNext 

Introduction

Overview

The release of the .NET Framework has changed the programming landscape for Microsoft application developers. Graphics programming has been affected as much as any other area. Whereas the Windows Graphical Device Interface (GDI) was once used to draw to a device, the .NET Framework now gives us *GDI+*.

GDI+ consists of a number of namespaces and classes, made available as part of the .NET Framework and designed specifically to allow developers to do custom drawing in web and Windows controls. Effectively acting as a wrapper around the old GDI, GDI+ provides an easy-to-understand, inheritance-based object model for use by .NET programmers.

The increased level of usability of GDI+ may draw many more programmers into custom drawing. Those from a C++/Microsoft Foundation Classes (MFC) background will already be familiar with many of the approaches in GDI+. Programmers who worked with Visual Basic will find in GDI+ an accessible way to develop controls that augment those shipped with the .NET Framework.

A substantial part of the .NET Framework base-class library falls under the GDI+ banner. This book concentrates on the areas and features of GDI+ that are relevant to programmers who are most likely to work with these classes; that is, programmers who want to develop *custom controls*.

 PreviousNext  PreviousNext 

Who Is This Book For?

This book is designed to appeal to any developer who wishes to appreciate and exploit the functionality of the GDI+ base classes provided as part of the .NET Framework. In particular, the book is aimed at those working in a business-application environment, writing GDI+ code for Windows Forms in order to create custom controls.

The examples in this book are presented in the C# language. They should be accessible to both intermediate-level C# programmers and relatively experienced developers capable of recognizing the standard constructs and features of an object-oriented language.

However, the choice of C# for the examples in this book is less important than the fact that it is a book about developing with GDI+ and custom controls in .NET. Indeed, the GDI+ techniques covered here are language-neutral. They can be applied, with the appropriate language knowledge, using other .NET Framework languages such as Visual Basic .NET and Visual C++ .NET.

 PreviousNext  PreviousNext 

What Does This Book Cover?

In this book, I'll cover GDI+ in three broad steps. First, I introduce GDI+ and its main features. Next, I tackle the nuts and bolts of GDI+ programming—the classes, methods, properties, and events that are most important to the application of GDI+ in custom controls, with plenty of description and demonstration. Then the chapters focus specifically on custom control architecture and development. I'll cover the issues you need to consider when deciding which features to build into your application, taking into account the requirements of the user interface.

Here is a summary of each chapter's contents:

- [Chapter 1](#): Here, I provide an overview of GDI+ and give some context for the development and use of custom controls in .NET applications.
- [Chapter 2](#): This chapter looks at GDI+ fundamentals such as drawing surfaces and coordinate systems. It includes an introduction to the `Graphics` object, which encapsulates a drawing surface. You'll see that there are basically three types of drawing surfaces: screens, bitmaps, and printers. The `Graphics` object provides methods for drawing and painting onto your drawing surface, as well as for managing many other features; for example, measuring units, transformations, clipping, and invalidation. Hence, the `Graphics` class provides the foundation for much of the rest of the book.
- [Chapter 3](#): This chapter covers pens and brushes. GDI+ makes a distinction between drawing and filling, and this fact is encapsulated well in the existence and juxtaposition of the `Pen` and `Brush` classes. These two classes are fundamental graphics tools, and you'll explore their versatility in this chapter.
- [Chapter 4](#): This chapter is about text and fonts. You'll see how GDI+ gives you programmatic control over the content and nature of the text in your graphics. You'll examine the `Font` and `FontFamily` classes, and understand how they relate to the rest of your graphics.
- [Chapter 5](#): The focus of this chapter is images and image manipulation. You'll explore the two GDI+ classes that allow you to handle images: `Image` and `Bitmap`, and even learn how to create and save images.
- [Chapter 6](#): This chapter introduces the important subjects of paths and regions, encapsulated in GDI+ by the `GraphicsPath` and `Region` classes. You'll see how these classes further highlight the distinction between drawing outlines and filling areas, enabling you to work with irregular shapes and to group sets of shapes together for more effective processing.
- [Chapter 7](#): This chapter expands on the use of the `GraphicsPath` and `Region` classes. You'll see how you can use these objects in operations such as clipping and invalidation, to create some useful and powerful effects.
- [Chapter 8](#): This chapter explains how to handle transformations in GDI+. Although translations, rotations, and shearing aren't commonly used in custom control development, this chapter will highlight some interesting techniques that may be appropriate in some applications.
- [Chapter 9](#): This chapter turns to the subject of printing. You'll develop an appreciation of how to control differences between what you see on the screen and what the printer outputs to the page.
- [Chapter 10](#): This chapter describes how to set up an alternative coordinate system, which you may find helpful in your development of custom controls. It involves subtle changes in the semantics of the `Graphics` class. You cannot inherit from the `Graphics` class itself (because it is defined as sealed), but you can "modify" its behavior by employing a design pattern called the decorator pattern, in which you define your own class whose methods have identical signatures to the methods in the `Graphics` class.
- [Chapter 11](#): This chapter provides in-depth coverage of custom controls, including their characteristics and how they compare with components. You'll see examples of custom controls that demonstrate their properties, focus, and events. You'll also learn how to derive a custom control from an existing Windows Forms control and how to develop composite custom controls.
- [Chapter 12](#): This chapter considers some of the ways to build in functionality for your custom control to make it easier for other developers to use your control in their code. It focuses on better integrating your controls with Visual Studio .NET, such as creating modal boxes and drop-down boxes to edit properties. All these features are intended to make life easier, at design time, for the developer using your control.
- [Chapter 13](#): This chapter looks at your options for implementing scrolling in your custom controls. You'll learn about the built-in support for scrolling in the Windows Forms classes, and then why it is preferable to implement scrolling through a custom control. You'll also discover how to implement smooth scrolling, to add a more polished feel to your application.
- [Chapter 14](#): This chapter is about another user-interaction feature: the mouse. You'll see how to develop your control so that it performs different functions when the mouse is used in different ways, including hit testing, mouse-event routing, dragging, and drag-scrolling. By their nature,

graphical user interfaces are visual, so you'll also take a look at cursors in your applications and what you can do with them.

At the back of the book, you'll find three appendixes with additional information related to working with GDI+ and the examples in this book. [Appendix A](#) covers dealing with assemblies and namespaces. [Appendix B](#) is about working with the console in Windows application. Finally, [Appendix C](#) looks at disposing of objects based on classes that implement the `IDisposable` interface.

 PreviousNext  PreviousNext 

What You Need to Run the Examples

To run the examples in this book, you'll need the following:

- A suitable operating system (at the time of writing, Microsoft recommends Windows XP Professional)
- The .NET Framework Software Development Kit (SDK)

The examples in this book are developed using the Visual Studio .NET integrated development environment (IDE). It is possible to build the examples *without* this IDE, but I don't cover that in the book.

I recommend that you also download the complete source code for the examples in this book, from www.apress.com, as described next. Whether you plan to cut and paste the code from these source files or type the code for yourself, the source code provides a valuable way to check for errors in your code.

 PreviousNext  PreviousNext 

Source Code and Errata

As you work through the examples in this book, you may choose either to type in all the code by hand or to use the source code that accompanies the book. Many readers prefer the former, because it's a good way to become familiar with the coding techniques.

Whether or not you want to type in the code, it's useful to have a copy of the source code handy. If you type in the code, you can use the source code to check the results you should be getting; it should be your first stop if you think you might have entered something incorrectly. And if you don't like typing, you'll definitely need to download the source code from the web site. Either way, the source code will help you with updates and debugging.

All the source code used in this book is available for download at www.apress.com. Simply click the Source Code link on this book's web page to navigate to where you can obtain the code. The downloadable source code files have been archived using WinZip. You'll need to extract the files you've downloaded to a folder on your hard drive, using a decompression program such as WinZip or PKUnzip. When you extract the files, the code will be extracted into chapter folders. When you start the extraction process, ensure that your decompression program is set to use folder names.

The publishing team and I have made every effort to make sure that there are no errors in the text or in the code. However, if you find an error in this book, like a spelling mistake or a faulty piece of code, we would be very grateful to hear about it. By sending in errata, you will be helping us provide even higher-quality information. To submit errata, simply use the Submit Errata link on the book's page on the Apress web site. We'll check the information, and (if appropriate) we'll post a message to the errata pages, and then use it in subsequent editions of the book.

 PreviousNext  PreviousNext 

Contact the Author

Please feel free to contact the author with questions and comments regarding the book. You can reach Eric at gdiplus@ericwhite.com.

 Previous

Next 

 PreviousNext 

Chapter 1: .NET Graphics Programming

Overview

Writing graphics code is one of the most enjoyable tasks in the computer programming arena. You may be building a custom graphical window that presents data in a more visible, more accessible fashion. You may be interactively creating graphics that are served to the user as part of a web site. Whatever you're doing in this area, the job of writing code to create colorful, effective graphics is an immensely pleasing and satisfying one.

GDI+ is the next-generation graphics device interface for Microsoft Windows operating systems. It is the future of graphics programming for Windows. Its object-oriented class library allows you to build many types of graphics applications.

One of the most powerful ways to use GDI+ is to build custom controls. Windows Forms comes with a stock of comprehensive controls, but there will certainly come a time when you need more. You can build your own custom controls that enhance the power and usability of your application.

This book focuses on the combination of GDI+ and custom controls, and specifically on the creation of custom controls using graphics in the .NET Framework and the C# language. We'll begin by looking at the GDI+ basics, before moving on to their application in the creation of useful controls. In this chapter, we'll look at two main areas:

- What GDI+ is—its relationship to the .NET Framework, its purpose, how it relates to previous graphics libraries, and a brief introduction to the namespaces and classes it encompasses.
- What custom controls are—an overview of Windows Forms and Web Forms custom controls, and how .NET helps you to create them.

So, let's start by examining GDI+ and the .NET Framework.

 PreviousNext  PreviousNext 

GDI+ and .NET

Most of you will already be aware of the many and varied reasons for adopting .NET, and will already be familiar with C#. But as a starting point, let's highlight two benefits of the .NET platform that are particularly relevant to this book.

Client Application Development with Windows Forms

.NET offers rich client application development via Windows Forms. Windows Forms is a very cleanly designed class hierarchy for building Windows applications. A highly interactive environment with properties and events, combined with a truly object-oriented approach, makes for an exceptionally powerful toolset. When you use it with the distributed computing features in the .NET Framework, you can build n-tier applications easily. The facilities for building custom controls are the best that I have ever seen.

Web Application Development with Web Forms

.NET allows web application development via ASP.NET and Web Forms. Together, ASP.NET and Web Forms provide a wonderful environment for building web sites. Many of the ideas and technologies in ASP.NET were previously evident in other environments, but there are also some completely new ideas. Because of the way that ASP.NET is built, you avoid the debugging issues that are problematic with other similar technologies. Among all the integrated development environments (IDEs) I've seen, Visual Studio .NET has the best features for improving programmer productivity.

.NET Features

.NET supplies an advanced set of features to help you create custom controls, including the following:

- **True integration of properties:** Properties are implemented in a Visual Basic-like style, as opposed to the "magic naming convention" used to implement properties in the Java world.
- **Effective event handling:** The inclusion of delegates as an integral part of the language provides a very good mechanism for writing event handlers.
- **Better design-time environment:** Visual Studio .NET gives you an unparalleled level of power for building custom controls. Not only can you seamlessly deploy custom controls in the Design window, but you also can explicitly control the appearance and manipulation of the controls and implement property editors in a variety of ways. The inclusion of attributes in the languages allows you to configure the design-time environment.
- **Powerful graphics device interface:** As you'll see in this book, using GDI+ to build custom controls gives you a broad and colorful palette with which you can "paint" your applications.
- **Effective infrastructure for distribution of custom controls:** After you've built your custom controls, you may want to distribute them to developers in other programming groups or sell them. You may also want to purchase commercial custom controls from others. The assembly infrastructure in .NET solves the problems of previous component distribution technologies.

We've seen some of these features in other development platforms before, but until now, we've never seen them all in a single platform. Their combination provides a powerful infrastructure for building custom controls.

 Previous

Next 

 Previous

Next 

What is GDI+?

GDI+ is Microsoft's new .NET Framework class library for graphical programming. Because it is part of the .NET Framework, it is object-oriented, of course. Organized into six namespaces, GDI+ is a set of classes that are designed to work together. In this way, it is similar to other areas of functionality in .NET, such as ADO.NET. Let's look at some of the major features of GDI+:

- GDI+ provides three types of drawing surfaces: windows, bitmaps, and printers.
- GDI+ provides tools that allow you to draw two-dimensional "line-drawings" on any drawing surface. These include facilities for drawing lines, many types of shapes and polygons, and curves, with a vast variety of brushes and pens. You have a number of types of transformations at your disposal, allowing you to create sophisticated effects with great ease.
- The text-drawing features in GDI+ are extensive, and the anti-aliasing technology is particularly impressive. *Anti-aliasing* is a technique to approve the appearance of graphics and text. We have all seen the "jaggies," or stairsteps, when drawing diagonal lines. Anti-aliasing reduces this effect.
- GDI+ has image and bitmap support. You can read images and draw them onto any drawing surface. You can also create images and draw within them.
- GDI+ supports printing. Print preview capabilities are obtained easily, with very little additional effort on the part of the developer.
- GDI+ is designed to work with any kind of .NET application.

Since GDI+ is part of the .NET Framework, and hence available to both Windows Forms and ASP.NET applications, you can (via an appropriate class design) write your custom graphical drawing code in such a way that you can use the *same* code in *both* types of applications. If you use the .NET Framework to write web services, you can wrap the same graphical code for use in those applications, and hence (for example) serve up custom images to clients of the web services that you write.

So, when building ASP.NET applications, you can use GDI+ to create images and serve them to the client's browser. When implementing a server farm, you can load-balance by distributing a

computationally intensive operation that creates graphical images over a number of computers or processors. Client applications often need to display graphics, either in custom controls or in forms. Web services can also make use of GDI+ to distribute graphical image generation among computers separated by a firewall or servers using technologies other than .NET.

In this book, we are not going to explore ASP.NET applications, nor building web services. We are going to focus solely on building Windows applications using Windows Forms.

To begin, let's place GDI+ in context and briefly consider the underlying technology: the Windows Graphical Device Interface (GDI).

 Previous

Next 

 Previous

Next 

GDI and GDI+

GDI+ is based on GDI, which is the part of the Windows Application Programming Interface (API) that handles graphics. In fact, GDI+ is effectively a "wrapper" around GDI (similar to the way that the Microsoft Foundation Classes, or MFC, wrapped GDI with its own classes). Perhaps someday Microsoft will implement GDI+ in a more native fashion, but for now, it is a wrapper around the Windows programming interface.

Note The term *GDI* refers to the graphics programming library that is part of the Windows API. The term *GDI+* refers to the managed code GDI+ class library that comes as part of the .NET Framework.

GDI offers a layer of abstraction, shielding the application developer from the need to program for each specific display device that the application may come across. Thus, when you want your application to draw to a screen, the application executes the appropriate GDI function, and then GDI works out how to communicate that to the video card. GDI is typically implemented through C or C++ programs (although many languages—including Visual Basic, C#, and Visual Basic .NET—can use GDI directly in much the same way that they are able to use many other parts of the Windows API).

While GDI+ sits above GDI, it offers two significant advantages. First, GDI+ adds capabilities to GDI by the inclusion of classes that implement functionality that would be difficult to write in straight GDI. To use such functionality in a GDI program, you would need to either acquire a library that supplies that functionality or write it yourself. For example, GDI+ provides specific enhancements for drawing semitransparently, access to gradient brushes, support for a wider range of image formats, easier programming of graphics paths and regions, and more powerful transformations. You'll encounter all of these during the course of this book. Second, GDI+ is fully integrated into the .NET Framework. GDI is a fairly complex, inaccessible technology. GDI+ has made the job of the programmer much easier.

The integration of GDI+ into the .NET Framework and its improved programming interface combine to bring other benefits:

- GDI+ uses an object-oriented class hierarchy. Programming GDI+ consists of creating objects, setting their properties, and calling their methods.
- GDI+ uses method overloading. Thus, when you call a method of a GDI+ object, you often have the opportunity to choose the version (or overload) of the method that takes the set of arguments most appropriate for the situation.
- GDI+ recognizes the distinction between drawing outlines and filling regions, and provides two separate operations for these tasks. However, GDI combines the two operations, and that leads to programmatic idiosyncrasies such as hollow brushes and hollow pens. If you've had experience with hollow pens and brushes in GDI, you know that they are a bit of a kluge. If you are never going to use GDI directly, and are always going to use GDI+, you don't need to worry about them; you will never see them. The result is a far cleaner programming interface.
- GDI+ hides some aspects of state management from the developer. GDI (not GDI+) keeps too much state, such as the notion of a current pen, current brush, and current point. Such artificial and unnecessary constraints are eliminated with GDI+. GDI+ shields the programmer from this complexity. This facilitates a simpler approach that eliminates the bugs that can occur when a developer doesn't completely set all necessary state (and thus "inherits" state randomly from other parts of the code).

For example, every line drawing method in GDI+ requires a `Pen` object. GDI+ knows what pen is currently part of the state of GDI, and if the pen passed as an argument to the GDI+ method is different from the pen currently selected in the device context, GDI+ automatically changes the pen in the device context. If the pen passed to the method is the same as that currently selected in the device context, GDI+ doesn't need to change the pen in the device context. GDI+ still maintains some state, such as transformations and clipping regions, but these can arguably be considered to be an attribute of the drawing surface, and should be maintained as state, whereas the notion of a current pen really is quite artificial.

GDI+ benefits from the perspective gained from the experience of building many graphical programming interfaces. It is a very clean, consistent, and powerful set of high-performance classes that provides most of the graphics functionality that you need. However, there are places where the functionality of GDI+ is limited, such as with bit block transfers or caret. Where GDI+ doesn't meet your needs, you can always drop down into the GDI API and use GDI in conjunction with GDI+. You'll see how to do this later in the book, when we discuss how to handle scrolling and mouse events in custom controls (scrolling is covered in [Chapter 13](#), and mouse events are discussed in [Chapter 14](#)).

WHAT ABOUT DIRECTX?

DirectX is another Microsoft technology related to graphics programming. However, DirectX differs significantly from GDI+ in both purpose and programming interface. DirectX is a suite of multimedia APIs that give applications the ability to access features of PC hardware (such as sound and graphics acceleration cards), enabling them to deliver high-performance output. Such high-quality presentation is commonly encountered in PC games where three-dimensional effects are a crucial part of the application.

GDI+ and DirectX are both libraries for writing graphical applications, but in our consideration of graphics for Windows and ASP.NET business applications, we need to concern ourselves only with GDI+. For all practical purposes, and for the purposes of this book, the two technologies are mutually exclusive.

The GDI+ Namespaces

As I mentioned earlier, all of the GDI+ functionality is contained in six namespaces that are included in the .NET Framework. [Table 1-1](#) describes the six GDI+ namespaces.

Table 1-1: GDI+ Namespaces

Namespace	Description
<code>System.Drawing</code>	Provides basic graphics functionality, including drawing surfaces, images, colors, brushes, pens, and fonts.
<code>System.Drawing.Drawing2D</code>	Provides advanced raster and vector graphics functionality.
<code>System.Drawing.Imaging</code>	Provides advanced imaging functionality, over and above that provided in the <code>System.Drawing</code> namespace.
<code>System.Drawing.Printing</code>	Provides print and print preview functionality.
<code>System.Drawing.Text</code>	Provides advanced font functionality, over and above that provided in the <code>System.Drawing</code> namespace.
<code>System.Drawing.Design</code>	Provides functionality for enhancing design-time support of custom controls. Includes classes for developing custom <code>UITypeEditor</code> classes, which allow you to customize behavior of custom controls in the Design window of Visual Studio .NET.

All of these namespaces are contained in a DLL file called `System.Drawing.dll`. You'll meet many of the classes as you progress through the early chapters of this book.

Since the focus of the book is the use of GDI+ in the construction of custom controls, let's now take a closer look at what that will require.

 PreviousNext 

Custom Controls

You can use two fundamental types of custom controls in .NET:

- **Windows Forms custom controls:** These controls are used in rich client applications. These are typically highly responsive custom controls. For example, a Windows Form custom control may track the mouse when it enters the control, and it may change the mouse cursor or even the appearance of the control itself.
- **Web Forms custom controls:** These controls are used in web applications. Web Forms controls will typically be less dynamically interactive than Windows Forms custom controls.

This book will focus on Windows Forms custom controls.

Benefits of Custom Controls

Custom controls are one of the most powerful approaches to application development. A well-designed custom control adds to your toolbox a tool that you can reuse time and time again. With a number of such controls providing large parts of your application's functionality, much of your application construction can simply be a case of adding controls to your application and setting properties. Here are some examples:

- For a check reconciliation program, you might develop a specialized grid control that allows your users to enter checks in the most efficient manner.
- An accounting application might require the user to enter a general ledger account number in many different windows. You can develop a custom control that takes care of all of the specifics of entering general ledger account numbers, eliminating work whenever you need to use it.
- A program that plays music might require features such as play, stop, rewind, and fastforward buttons, as well as a slider for setting the volume.

These are all cases where you might employ GDI+ in the construction of custom controls. Thus, when you bring these two technologies together, you have quite an animal. When you employ graphics code in your custom controls, you have an enjoyable way to improve programmer productivity, while simultaneously increasing the usability and visual appeal of your applications. By building a variety of custom controls, you can affect the entire appearance and usability of your application. You can add custom controls that fit in with the standard Windows controls, or you can supplant the look and feel of the entire set of Windows controls with your own controls, and thus create a game-like environment that is appealing to an entirely different audience.

Types of Windows Forms Custom Controls

Within Windows Forms custom controls are three basic types of custom controls, each requiring a different programming approach and used for a different purpose:

- A custom control can use GDI+ to draw itself.
- A custom control can be derived from another control, inheriting most of its functionality from the base class.
- You can compose a custom control from several existing controls, and use them as a group.

Note You can use GDI+ in your application without building a custom control, but later, you'll see that it is often better to build a custom control than simply to draw in a Form or Panel.

Let's look at the mechanics and functionality of custom controls.

.NET Facilities for Building Windows Forms Custom Controls

You'll find extensive facilities in the .NET Framework for building Windows Forms custom controls. The Framework defines how you should build your custom control, and if you follow certain conventions, your custom control does not need to be just a stand-alone control in a window—it can be used in conjunction

with other Windows Forms controls. You can tab between the custom control and the standard controls, and the user can tell when your custom control has the focus.

When you build custom controls in Windows Forms, you derive from the `Control` class, the `UserControl` class, or any existing control or previously developed custom control. Then you can write code that responds to `Paint` events and draws whatever graphics you want. Using GDI+, perhaps in conjunction with other controls, you can thus create the look of your custom control.

Inside your control, you may build code to respond to keyboard and mouse events. By following certain conventions, your custom control can participate in the user's navigation among other controls, both native controls and other custom controls. Your keyboard and mouse event handlers, perhaps (again) in conjunction with other controls, combine to create the feel of your custom control. You can add properties, methods, and events to your custom control as you develop it.

Visual Studio .NET Support for Custom Controls

Again, by following certain conventions, Visual Studio can host your control at design-time. At the most basic level, this means you'll be able to do the following:

- Modify the layout of your custom control in the Design window.
- Edit the properties and events (both the events that you defined and those that the control inherited from its base class) of your custom control in the Properties window.
- Add your custom control to the Toolbox window for dragging and dropping into the Design window as appropriate.

You can enhance your custom control so that it participates even more fully in the Visual Studio .NET IDE. There are two separate modes (runtime and design-time) in which the custom control needs to operate. Runtime mode is the most obvious mode in which you run your custom control. In runtime mode, the custom control performs its intended functions. More interestingly, custom controls can also function in a design-time mode (seen when the control is viewed in the Design window).

As an example, consider an elaborate multicolumn grid control. When you run it in the application, you'll see the data with which you populate the grid. However, when you run it in the Design window, you might not see the actual data, but instead see the name of the database table and column that supplies the data. In the application, you can click cells and change the data. In the Design window, you can right-click the grid and change the source of the data. This represents a significant benefit to developers.

Often, in other environments, the specialized code that renders the custom control in the Design window mode is embedded in the IDE itself. This means that if you develop new, elaborate custom controls, you don't have the capability to change the IDE so that your new custom control can allow the developer to manipulate the setup of the control. The IDE usually has some default behavior for modifying the setup of the control. You typically can only set properties to configure the custom control. You cannot modify the design-time implementation of the control to allow for such things as right-clicking to change the data source. This means that when you develop new, elaborate custom controls, in the IDE, you often see only a rectangle that represents the position of the custom control.

However, in Visual Studio .NET, all such design-time rendering and interaction are separate from the implementation of the IDE. You can add code to your custom control that enhances the design-time support, and you can write additional classes that completely customize the appearance and developer interaction of the custom control within the design-time environment. In fact, the controls that come with the .NET Framework use exactly the same mechanism that is available to you when building your own custom controls.

 Previous

Next 

 Previous

Next 

What You Need to Know

This book covers some advanced topics. In general, a developer who is new to using Visual Studio and C# should read other books before reading this book. Therefore, this book makes the assumption of some basic skills on your part.

You should know how to create new projects in Visual Studio, using the Windows Application template. This book does not assume that you know how to create a Windows Control application. In [Chapter 11](#), we will go through the process of creating an application using the Windows Control Library template.

Once you have created the project, you should know how to manipulate various parts of the project. For instance, you should be able to set and examine properties in the Properties window. You should be able to create event handlers and delete them. In [Chapter 11](#), when we discuss generation of events, I'll provide a quick overview of delegates and events from an event generation perspective, but a basic knowledge of event handlers and writing them is assumed.

One of the subjects that can be vexing for programmers new to .NET programming is that of namespaces and assemblies. As you've learned in this chapter, there are six namespaces associated with GDI+. In addition, when creating classes, by default, the class is not created with all necessary references. You should be competent in managing namespaces and assemblies, including being able to understand the error messages associated with them and correct them by adding appropriate references and using statements. However, if you need a refresher in this area, [Appendix A](#) discusses dealing with assemblies and namespaces.

Debugging graphics code can be problematic, particularly when dealing with issues associated with event ordering. The debugger itself can alter event order. Putting up note boxes is no better, since the introduction of a modal dialog box also impacts event order. An alternative is to use the `Console` class from the .NET Framework. This class allows you to output text in such a way that you can easily see it, but it doesn't impact your event ordering. One of the peculiarities about the `Console` class is that it operates slightly differently when running in debug mode. In debug mode, output shows up in a window in Visual Studio. Otherwise, output shows up in a command window. A number of the examples in the coming chapters use the `Console` class. If you have any questions about using this class, see [Appendix B](#) for an explanation and a tutorial.

Finally, an anachronistic artifact of Microsoft Windows programming is the existence of special memory within the Windows kernel called *Resources*. Certain parts of Windows programming use this memory, and it must be released in a timely fashion; otherwise, Windows runs out of this memory and Bad Things Happen. The actual dynamics of these Bad Things changes from version to version of Windows. Things are getting better, but the problem hasn't been completely eliminated, even for the latest versions of Windows. This issue is particularly relevant to GDI+ programming, because many of the objects in GDI+ use this special memory. To solve this problem in a managed environment with garbage disposal, many classes implement the `IDisposable` interface. To make a long story short, you must call the `Dispose` method at the correct time on objects of such classes. But the story is longer than this. If you are not fully familiar with the issues of dealing with classes that implement `IDisposable`, [Appendix C](#) tells you what you need to know.

Previous

Next

Previous

Next

Summary

This chapter provided an overview of GDI+ and examined the use of custom controls in the .NET Framework.

You learned that GDI+ is based on GDI. It is a layer over GDI, but adds a lot of functionality to GDI. So, while you'll usually take advantage of the new, improved programming interface for graphical operations that GDI+ offers, you can always use GDI directly when GDI+ doesn't meet your needs.

Next, we looked at custom control design and implementation in .NET. The chapter identified three key features that make custom control technology powerful: the ability to produce controls with defined properties, defined events, and a customizable design-time appearance.

Then I noted four areas that you should be familiar with before proceeding with the rest of the book:

- Managing properties and event handlers in Visual Studio
- Configuring projects to include various assemblies and namespaces
- Using the `Console` class

- Correctly using classes that implement `IDisposable`

In the [next chapter](#), you will learn about the drawing surfaces related to the target environments for the graphics resulting from GDI+ coding.

 Previous

Next 

 PreviousNext 

Chapter 2: Drawing Surfaces

Overview

When writing GDI+ code, you have three basic target environments for the resulting graphics: a window on a screen (a *form*), a *page* being sent to a printer, or a bitmap (an *image*) in memory. Each environment presents a *drawing surface*—a pixel-based representation of the form, page, or image. The defining characteristics of each drawing surface are the same (size, pixel resolution, and color depth), but the manifestation of these characteristics and the way in which you can control them are different in each case. In this chapter, we will explore these drawing surfaces and their differences, so that you can produce correct and efficient code for each environment.

The *coordinate system* is used to describe the points of the drawing surface. Through the coordinate system, you define the size, shape, and position of each element of your graphic. When writing GDI+ code to produce complex and intricate graphics, and particularly when producing custom controls, you will find that a single misplaced pixel can greatly impair the visual impact of your work. Tests have shown that humans can see an object as small as 1/3000 inch across when the object is a light source. In comparison, a pixel on a very, very high-resolution screen is approximately 1/180 inch square, and most screens have much larger pixels. Individual pixels are very visible! By gaining an understanding of the coordinate system of the surface (and some related issues, such as anti-aliasing), you will be able to determine exactly which pixels will be affected when you invoke a drawing operation in your code. Consequently, you will be able to produce precise and effective graphics.

This chapter starts with a comparison of raster-based and vector-based systems, to help you understand how the raster-based drawing surfaces work. Next, it covers the drawing surface characteristics and their singularities in each environment, with particular emphasis on the use and definition of color in GDI+. Then it introduces the `Graphics` class, which encapsulates your drawing surface in each environment. Finally, we'll investigate the coordinate system implemented in GDI+ and review some of its limitations. These subjects form the foundation of any study of GDI+, so you will be ready to study the `Pen` and `Brush` classes in [Chapter 3](#), and indeed, all the subjects in subsequent chapters.

 PreviousNext  PreviousNext 

Raster-Based vs. Vector-Based Systems

A *drawing surface* is a raster-based abstraction of a window on a screen, or a page on a printer, or an image in memory. *Raster-based* means that it is composed of pixels arranged in a two-dimensional grid. A *pixel* is the smallest element of a drawing surface that you can directly manipulate, and you address individual pixels by using a pair of coordinates in the form (x, y) . These are generally called *Cartesian coordinates*.

Note The word *pixel* was originally derived from the term *picture element*.

In order to appreciate the characteristics of a raster-based system, it helps to compare it to other systems that are not raster-based.

When you draw a line in a *raster-based* system, the overriding characteristic is that the line is represented by the coloring of certain pixels in the grid. You can define the line by expressing exactly which pixels are colored. In a *vector-based* system, the action of drawing a line is quite different. In this case, the line is represented as an entity that starts at a well-defined point, has a specified length, travels in a specified direction (specified by an angle or path), and is infinitely thin.

For example, [Figure 2-1](#) shows a representation of a vector-based graphic, which includes a couple of straight (infinitely thin) lines and a curve. The same image on a raster-based display might look something like [Figure 2-2](#).

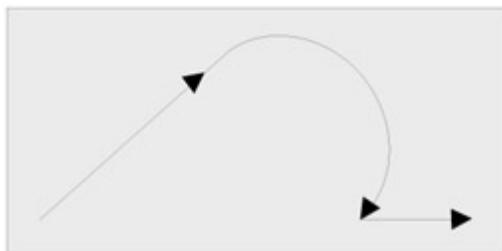


Figure 2-1: A vector-based graphic

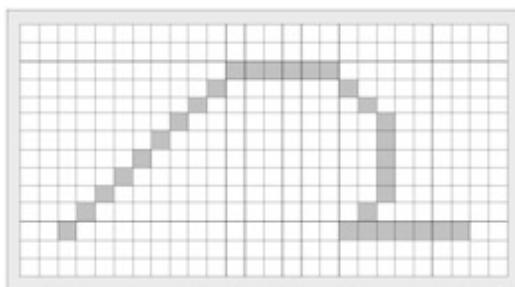


Figure 2-2: A raster-based graphic

The effectiveness of the conversion from vectors to pixels depends on the *pixel resolution* of the raster-based drawing surface; that is, the size of the pixels. In this case, the image shown in [Figure 2-2](#) is very low-resolution. In that image, the pixels are relatively large, and so the image is not a high-quality representation of the original vector representation ([Figure 2-1](#)). But this example serves to illustrate the difference between vector-based and raster-based systems.

These days, almost all devices are ultimately raster-based. Raster-based devices are far easier to engineer and manufacture. Often, we pretend for programming purposes that they are vector-based, because it's easier to describe our image using vector-based techniques and to depend on some conversion mechanism to turn it into a raster-based representation at the time we wish to project the image description to a drawing surface. When we are writing some graphics code, we want to think in terms of "start the line at location A and draw to location B, with a given thickness." We don't want to think about individual pixels. The pixels don't match our conceptual models, whereas vector-based descriptions do.

A few years ago, vector-based displays were more common than they are now. Much time could be spent defining each vector programmatically for a vector-based display. Complicated electronics in the cathode ray tube (CRT) would direct the beam in such a way that it would draw the vectors that the programmer specified. Anyone who has played the original Battlezone or Asteroids arcade machines has witnessed vector graphics firsthand. Large, pen-based plotters were also fundamentally vector-based.

But the important technique used today is to *write* graphics code primarily using vector coordinates, and then have these vectors *displayed* on a raster-based device. Indeed, many of the drawing operations in GDI+ are expressed in terms of vectors. GDI+ then takes the vector expression and renders it to the raster-based drawing surface.

Previous

Next

Previous

Next

Characteristics of a Drawing Surface

Any drawing has three main characteristics: its size (width and height), its pixel resolution, and its color depth. These characteristics play a key role in determining how your method calls affect the drawing surface itself. In addition, these characteristics play a part when you draw first to an `Image`, which is a drawing surface, and then draw that `Image` to either a `Form` or a page on a printer. For example, if you have a very high-resolution bitmap, and you draw it on a low-resolution drawing surface, the bitmap will be scaled and therefore appear much larger.

Let's examine each of these characteristics a little more closely, beginning with the simplest one: size.

The Size of a Drawing Surface

The size of a drawing surface is defined by the number of pixels horizontally and vertically that can be addressed by the drawing surface. For example, the raster-based image shown earlier in [Figure 2-2](#) is drawn on a grid of 26 pixels horizontally by 14 pixels vertically.

You can query the drawing surface to determine its size, before you begin drawing (through the `x` and `y` properties of the `ClientRectangle` of the drawing surface, which you'll meet later in this chapter).

Sometimes you can change the size directly, and sometimes you can't. This depends on the type of the drawing surface, as you'll learn in the "[Drawing Surfaces in Different Environments](#)" section later in this chapter.

The Resolution of a Drawing Surface

In GDI+, the resolution of your drawing surface is always expressed in units of *pixels per inch* (PPI) or *dots per inch* (DPI). If a surface has a resolution of 72 DPI, you should be able to examine a 1-inch square of the surface and find that it is composed of a grid of 72 pixels in the horizontal direction by 72 pixels in the vertical direction. (It seems a little arbitrary that we don't have an equivalent way to specify resolution in metric terms, but it is understandable, in that it would complicate the programming interface significantly.)

Note These days, pixels are square, and so the resolution is the same in both the horizontal and vertical directions. There was a period of time back in the 1980s when some screen devices used rectangular pixels, and this was a bothersome situation.

Sometimes you might want to find out the resolution of the drawing surface. If you were drawing a complicated engineering diagram, you might take advantage of a high-resolution printer to produce the best image possible. But if you were drawing to a low-resolution surface such as a form on a window, you might alter your drawing algorithm to produce an image that looks great on a screen. In GDI+, you can always call a method to find out the resolution of the drawing surface.

Screen Resolution

The screen is assumed to have a resolution of 96 DPI, regardless of the *actual* resolution of the screen. This is because historically, there has been no way to query the graphics card driver and determine the *actual* resolution of the screen; indeed, in the past, the graphics driver didn't even know the screen's resolution. A screen of any size can be hooked up to an analog VGA output, and the user can adjust the settings of the physical screen device to change the DPI, but the graphics card is not aware of these adjustments.

This situation may change with support of more and more plug-and-play monitors that inform the graphics card driver of their size. Knowing the size of the monitor and the current dimensions of the display (1024 X 768, 800 X 600, and so on) provides enough information to be able to calculate the resolution.

Printer Resolution

Every printer device also has a resolution. Users can select their desired resolution. Alternatively, you can force resolution programmatically at the time of printing. We'll investigate this subject in more detail in [Chapter 9](#).

The Color Depth of a Drawing Surface

Many operations in GDI+ require a color. Whenever you draw a line, a rectangle, text, and so on, GDI+ needs to know what color you want to use for that drawing operation. The *color depth* of a drawing surface defines how many colors can be stored in each pixel.

Color Support

In GDI+, each pixel describes a color that is made up of a combination of red, green, and blue (RGB) components. There are 256 levels of intensity for each of these three constituent colors (represented by values in the range 0 to 255), and thus each pixel can contain one of $256 \times 256 \times 256 = 16,777,216$ possible colors. Colors have 256 levels of intensity because each color is represented by 1 byte.

In order to describe all of these colors, the drawing surface must support a 24-bit color value (this is a

true color system). If it does, the colors are rendered on the drawing surface as you specify them in your program. However, some display systems do not support 24-bit color. They may use only 16-bit color (65,546 colors) or 8-bit color (256 colors). If you're drawing to a printer, it might be that the printer does not support color at all, or that it doesn't support a 24-bit color depth. And you can create images with a wide variety of color depths, too. For example, you can create monochrome images (in which only black and white are supported) and 8-bit images (in which only 256 colors are supported).

Dithering

In GDI+, the default drawing model uses 24-bit color depth. If the drawing surface doesn't allow for 24-bit color depth, then GDI+ sometimes restricts the types of graphical operations that can be done, and sometimes it takes care of the incompatibility by translating colors—a process known as *dithering*. In the dithering process, GDI+ uses alternating colors in adjacent pixels to give the closest possible approximation to the color that the display is not capable of rendering. [Figure 2-3](#) shows a highly magnified view of a gradient that is dithered.



Figure 2-3: A dithered gradient

Dithering is not a pleasing effect. To avoid dithering, you can call a method of the `Graphics` class, `GetNearestColor`, which will return a color that the display is capable of rendering without dithering. In the future, this issue will be less of a problem, since even the cheapest computers now have the capability to display 24-bit color depth.

[Previous](#)

[Next](#)

[Previous](#)

[Next](#)

The Color Structure

In GDI+, colors are encapsulated in the `Color` structure, which you can employ in three ways: by using RGB values, by referencing predefined colors, or by using HSB values. GDI+ colors also have an alpha component.

RGB Values

You can create a new instance of the `Color` structure by passing RGB values into a static function in the `Color` structure, like this:

```
Color c = Color.FromArgb(100, 100, 255);  
Brush b = new SolidBrush(c);  
g.FillRectangle(b, ClientRectangle);
```

Thus, you specify the exact color by defining the intensity of each of the three components of the RGB scheme. As mentioned previously, the default drawing model uses 24-bit color, and the values passed into the three arguments of the `FromArgb` method must be between 0 and 255.

Predefined Colors

Another way to specify colors is by name. There are 141 predefined colors that you can reference by name. You use these colors by accessing public properties in the `Color` structure, as follows:

```
Color c1 = Color.LavenderBlush;  
Color c2 = Color.Red;
```

HSB Values

The third way to specify a color is to break it down into the following three components, which are

collectively known as the *HSB model*:

- **Hue:** This is the actual color, based on its wavelength. Hue is defined by a floating-point value between 0 and 360. This represents an angle in degrees, which, in turn, represents a color taken from a color wheel.
- Note** A *color wheel* is a disc that contains all possible hues. There is a one-to-one relationship between the colors of the visible spectrum and the radius lines of the disc.
- **Saturation:** This is also known as color intensity. A high-saturation color will appear to be very pure. A low-saturation color will appear to be washed-out. The saturation level is defined by a floating-point number, with a value between 0 and 1. The saturation level has an inverse relationship with the purity of the hue. The more other colors "interfere," the more the color will tend towards white or gray and thus appear washed-out.
- **Brightness:** This is the relative lightness or darkness of a color, expressed as a floating-point number between 0 (black) and 1 (white).

The `Color` structure contains static methods that allow you to get the HSB components: `GetHue`, `GetSaturation`, and `GetBrightness`. The following shows how you could use these methods:

```
Color c = Color.Blue;
float h = c.GetHue();
float s = c.GetSaturation();
float b = c.GetBrightness();
```

The HSB model of colors is not used much in programming. It is more common in graphic design and other such disciplines. When making custom controls, you primarily use the colors supplied by the system preferences set by the user.

The Alpha Component

Colors in GDI+ have an additional component, called the *alpha component*, which you can use to control the transparency or opacity of a color. This allows you to create fade-in/fade-out effects (such as the fade transition effect for menus and tool tips in Windows 2000 and later), and watermark effects that you can draw over top of other graphics.

The alpha component is specified by an integer value between 0 and 255, with 0 being completely transparent and 255 being completely opaque. If you omit the alpha value, GDI+ assumes you want the default value, which is 255.

All colors in GDI+ always have the alpha component. Often, it is set to 255, so the color is completely opaque. When you're working on graphical drawing that doesn't have any aspect of transparency, you typically ignore the alpha component of colors; you allow the color to be completely opaque, which is the default behavior.

Different drawing surfaces use the alpha component differently. The alpha component has an effect on the drawing operation to give the appearance of transparency, but the resulting rendition on the screen doesn't have any alpha component. The same is true for printer pages. However, it is a different story for images. Images themselves can contain an alpha component for each and every pixel in the image. There are ways that you can use alpha values to blend colors when drawing to an image, and there are ways that you can simply transfer the alpha value of the color to the image, so that the image retains the alpha component. Subsequently, when the image is drawn, the image will be drawn semitransparently, based on the value of the alpha component of the colors of the pixels.

The following is an example that demonstrates how you might use the `Color` structure and alpha component. First, we create two `Color` instances from the `Color` structure: `c1`, which is blue and has an alpha level of 100, and `c2`, which is green and has an alpha level of 50. Then we draw three filled shapes.

```
private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create two color instances with different alpha components
    Color c1 = Color.FromArgb(100, Color.Blue);
    Color c2 = Color.FromArgb(50, Color.Green);
```

```
Color c2 = Color.FromArgb(50, Color.Green);

// Now draw a red opaque ellipse
g.FillEllipse(Brushes.Red, 20, 20, 80, 80);

// Now draw a rectangle, filled with the semitransparent blue
g.FillRectangle(new SolidBrush(c1), 60, 80, 60, 60);

// Now draw a polygon, filled with the almost-transparent green
Point[] pa = new Point[] {
    new Point(150, 40),
    new Point(90, 40),
    new Point(90, 120)};
g.FillPolygon(new SolidBrush(c2), pa);
}
```

The first shape is an opaque red circular ellipse. The second is a semitransparent blue rectangle. Note that the rectangle overlaps the ellipse, and it is drawn *after* the ellipse (so that the rectangle appears to be layered over the ellipse). Because the rectangle is semitransparent (with alpha 100), you can see the edge of the ellipse through the rectangle. Finally, we draw a three-sided polygon (a triangle), filled with the almost-transparent green. Again, the overlapping and layering arrangement, combined with the near-transparency of the triangle, means that you can clearly see the parts of the rectangle and ellipse that lie underneath the circle, as shown in [Figure 2-4](#).



Figure 2-4: Shapes drawn semitransparently

The subject of colors is complex. The details presented here will be sufficient for building all but the most exotic of custom controls.

Previous

Next

Previous

Next

Drawing Surfaces in Different Environments

The three characteristics of the drawing surface described earlier—size, resolution, and color depth—exist in all three types of drawing surfaces—form, printer, and image—but there are significant differences in how the three types of drawing surfaces deal with these characteristics. This section provides a brief overview of all three characteristics for each type of drawing surface. You'll deal with these characteristics and surfaces in the upcoming chapters.

Drawing Windows Forms

A *form* (a window on a screen) is made up of client and nonclient areas. The border of the window and the controls that you use to manipulate the window itself (to resize it, minimize it, maximize it, close it, move it, or bring up the window menu) are called the *window decorations*. The size of the window includes the space necessary to display these decorations. The size of the client area includes only the area in which you can draw. The size of the drawing surface for a Form is the size of the client area of the window, which is generally not the same as the size of the window itself. [Figure 2-5](#) illustrates the

client area and typical decorations of a window.

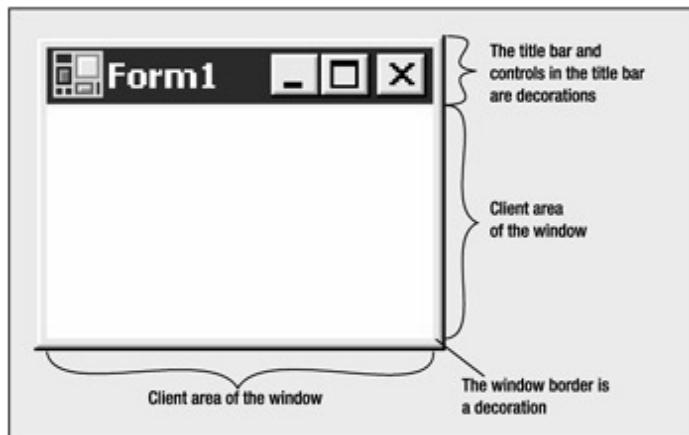


Figure 2-5: Parts of a window

In [Figure 2-5](#), the decorations include the title bar (and all of the buttons on the title bar) and the border that surrounds the window. It is possible to create a borderless form that would allow you to draw over the entire surface of the window.

The ClientSize Property

When you create a new form, either programmatically or using the Design window in Visual Studio, you specify the form's `Size` property, which sets the size of the entire window (including the decorations). Immediately after creating the window, you can set the size of the actual drawing surface using its `ClientSize` property. To change the drawing surface size for a form, use code like this:

```
this.ClientSize = new Size(120, 120);
```

For some types of forms, you might want to do some calculations during the `Load` event and determine what size you want the form to be. As an example, you might want these calculations to take into consideration the data that will be displayed in the window. Furthermore, the font selected for display in the form might also impact its size.

If you set the client size during the `Load` event, the user will not see the window resize because it effectively resets the window size *before* its `Paint` event comes through. In contrast, if you include the line in the code within another event, the user will see the window appear at its initial size, and then resize when that event fires. Note that the client size is given in the form of a `Size` object.

The ClientRectangle Property

Every form has a `ClientRectangle` property, which represents the window's client rectangle (the white area in [Figure 2-5](#)). For example, you can use the `ClientRectangle` property to fill the drawing surface with a solid color:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
}
```

You'll see this property often in the examples in this book.

Color Depth and Resolution in Forms

Developers have little control over the *color depth* of forms. Users set the color depth of their screen in their display settings, and the form inherits the color depth of the screen.

Screens primarily come in two varieties: CRTs and liquid crystal displays (LCDs). Both can have a variety of *pixel resolutions*, as well as a variety of sizes. As previously mentioned, there is currently no good mechanism for querying the type of hardware being used, and hence determine the *actual* pixel

resolution. GDI+ simply defines the pixel resolution for all display devices (including monitors of all sizes, and regardless of the screen resolution set in the display properties) as 96 DPI. This has repercussions when you are drawing images (which can have a variety of DPIs) to a form.

As an example, if you are displaying a high-resolution image on a low-resolution drawing surface, the image will be downsampled and be displayed in a small amount of real estate. Using this technique, you could create a thumbnail of an image. If you are displaying a low-resolution image on a higher-resolution drawing surface, the image will be enlarged, as if you were zooming in on the image. We'll examine this interaction in [Chapter 5](#).

Drawing Pages for a Printer

When it comes to printers, you cannot programmatically control the size of the drawing surface directly, but you can control it indirectly by changing the paper size or the printer resolution, as you'll see in [Chapter 9](#). In addition, the user can change these characteristics before printing by using a printer setup dialog box.

Although it may not be obvious, the drawing surface of a printer is, in fact, closely related to the drawing surface of a *print preview window*. This is intentionally implemented within the .NET Framework. The print preview window is an approximation of a printed page. It has the same aspect ratio and should look basically the same. Consequently, you program for these two drawing surfaces in an almost identical manner. The only difference is that each of these two drawing surfaces requires a different method to create it. Note that if the user sets a different paper size for the printer, the new drawing surface size is also reflected in the print preview window. You'll see how to set and query the drawing surface resolution and size for a printed page and for the print preview window in [Chapter 9](#).

Every printer has a default pixel resolution, but most printers provide a range of possible pixel resolution settings. A typical default pixel resolution is 300 DPI. Users can usually change their printer resolution through a page setup dialog box. In your code, you can query GDI+ programmatically to determine the printer's resolution. Programmatically, the print preview drawing surface has the same pixel resolution as the current default printer.

As usual, you program with the assumption that the printer and print preview window have 24-bit color depths. Users can set the color depth of their printer through its advanced properties. In addition, you can control color depth programmatically to a certain extent with the printer. If the target devices do not support the color depth, GDI+ works in conjunction with the printer driver to make the appropriate color mapping. You'll see how this works in [Chapter 9](#).

Drawing into Images

The size of the image represented by a GDI+ `Image` object (that is, the size of the drawing surface) is specified when the `Image` object is created. If you create the `Image` explicitly, then you specify the `Image` size when invoking the `Image` class's constructor. By contrast, if you read the image from a file (in a format such as BMP, GIF, or JPG), stream, or other source into an `Image` object, the size is specified by the source itself.

You can explicitly set the pixel resolution of an `Image` object to any desired DPI. If you read the image from a file, stream, or similar source, the resolution of the image is determined by the source. In [Chapter 5](#), you'll see how you can use an image editor to create an image with a specific DPI, and then read the image file into an `Image` object within your C# program and query it to check the DPI setting of the image file. You can also change the DPI of an `Image` object in memory, at any point after it has been created. The DPI of an image affects the drawing behavior, particularly when you try to draw the image to drawing surfaces with the same or different DPIs. We'll examine all of this in [Chapter 5](#).

`Image` objects have a wide variety of color depths, which vary from 1-bit for a monochrome bitmap, to 16-bit, 24-bit, 32-bit, 48-bit, or 64-bit depth. In general, the greater the color depth, the better the image quality. If the image has less color depth, it will be smaller, which impacts storage size and image transmission time over the Internet. Alternatively, the pixel color for the `Image` object can be set via a color-indexed value, which means that it uses values that are indexed into colors in the system color table. `Image` objects can also include an alpha component in the color definition. This means that you can make portions of an image completely or partially transparent.

You set the color depth of an image at the time of image creation. It's important to understand, however, that when you save an image to a particular format (such as BMP, JPG, or GIF), the .NET Framework may change the color depth of the image as it is saved.

Summary of Drawing Surfaces

[Table 2-1](#) summarizes the four types of drawing surfaces and the characteristics of image size, resolution, and color depth. I've included the print preview as a separate surface, although in many ways, it's related to the printer surface.

Table 2-1: Drawing Surfaces Summary

Surface	Image Size	Resolution	Color Depth
Forms	The size comes from the size of the client area of the window.	The resolution is 96 DPI.	Color depth may or may not be 24-bit. GDI+ renders to the display using dithering if necessary.
Printers	The size comes from the printer driver. You can change the drawing surface size by changing the page size of the printer, if this is supported by the printer.	The resolution comes from the printer driver. You can change this if the printer supports different resolution values.	Color depth may or may not be 24-bit. GDI+ and the printer driver are responsible for rendering to the printer.
Print Preview	You can change the image size by changing the page size.	The resolution is based on the DPI of the printer.	Color depth is the same as for forms. GDI+ renders to the display using dithering if necessary.
Images	The size is specified on creation of the image. If you read the image from a file, the image size is also read from the file.	If you create an image programmatically, resolution is 96 DPI by default. If you read the image from a file, resolution is also read from the file. You can change the resolution at any time.	Images can have a wide variety of color depths, including monochrome, grayscale, indexed into a color map, and 16-bit to 64-bit depth. In addition, alpha component for colors.

That completes our investigation of the different drawing surfaces presented in each environment. Before you can draw to any drawing surface, you must get an instance of the `Graphics` class, which is discussed in the [next section](#).

 Previous

Next 

 Previous

Next 

The Graphics Class

Any time that you wish to draw to a drawing surface, you must get an instance of the `Graphics` class. The manner in which you obtain your `Graphics` object will vary depending on your target environment (form, printer, or image).

Creating a Graphics Object for a Form

If you are drawing to a form in response to a `Paint` event, you will be handed a `Graphics` object—it is part of the `PaintEventArgs` argument of your event handler:

```
private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    ...
}
```

If you are responding to some other event, perhaps to a keyboard or mouse event, you can get a `Graphics` object from the form by calling the `CreateGraphics` method.

```
private void Form1_MouseMove(object sender,
                             System.Windows.Forms.MouseEventArgs e)
{
    Graphics g = this.CreateGraphics();
    // 'this' refers to an object of type Control
    ...
}
```

Although you usually would not want to create a `Graphics` object in this fashion, it may be suitable under certain circumstances. We'll explore this topic in [Chapter 14](#).

Creating a Graphics Object for a Printer or Image

When printing or opening a print preview window, GDI+ again hands `Graphics` objects to an event handler, similar to when you acquire a `Graphics` object in response to a `Paint` event. We'll go into the details on how to implement printing in [Chapter 9](#).

If you have an `Image` object, you can get a `Graphics` object from it by calling the static method `Graphics.FromImage`. Then you can draw to the image by drawing to this `Graphics` object.

Drawing onto a Drawing Surface

You draw onto a drawing surface by calling methods of your `Graphics` object. To give you an idea of the types of things you can do, [Table 2-2](#) lists a selection of the drawing and filling methods provided by the `Graphics` class. There are some variations on the methods listed here, but this table gives a good representation of the drawing capabilities of the `Graphics` class.

Table 2-2: Some Graphics Class Drawing Methods

Method	Description
Clear	Fills the entire drawing surface with a background color, which is specified in the method call.
DrawArc	Draws an elliptical arc. Thus, the arc is a portion of an ellipse. You specify a <code>Pen</code> object, the containing rectangle of the ellipse, and the start and sweep angles.
DrawBezier	Draws a Bézier spline. This is a curve drawn between two specified endpoints, whose curvature is controlled by two more specified control points. Thus, you specify a <code>Pen</code> object, two endpoints, and two control points.
DrawCurve	Draws a cardinal spline. This is like drawing a curve that is created by taking a strip of a flexible material and making it pass through a set of fixed points (before the advent of computer-aided design systems, architects and engineers used flexible strips to achieve this in their design drawings). Note that the path created in this way depends in part on the "flexibility" of the material; a more flexible material produces tighter bends. This property is also known as the <i>tension</i> .
DrawEllipse	Draws an ellipse. You must specify a <code>Pen</code> object and the ellipse's containing rectangle.
DrawIcon	Draws the image in the specified <code>Icon</code> object.
DrawImage	Draws all or part of the specified <code>Image</code> object to the specified area of the drawing surface.
DrawLine	Draws a straight line, using a specified <code>Pen</code> object, between two specified points.
DrawPath	Uses a specified <code>Pen</code> object to draw a path represented by a <code>GraphicsPath</code> object. We'll explore the <code>GraphicsPath</code> class in Chapter 6 .

DrawPie	Draws a pie shape. This requires a <code>Pen</code> object, a containing rectangle, and two angles, which indicate the arc of the pie.
DrawPolygon	Uses a specified <code>Pen</code> object to draw a polygon, which is represented by an array of points.
DrawRectangle	Uses a specified <code>Pen</code> object to draw the specified rectangle.
DrawString	Draws the specified text string in a specified font, within the specified containing rectangle. This method is overloaded to provide some optional decorative effects.
FillClosedCurve	Uses the specified <code>Brush</code> object to fill the interior of the specified closed cardinal spline curve.
FillEllipse	Uses the specified <code>Brush</code> object to fill the interior of the specified ellipse.
FillPath	Uses the specified <code>Brush</code> object to fill a path represented by a <code>GraphicsPath</code> object.
FillPie	Uses the specified <code>Brush</code> object to fill the interior of a pie shape.
FillPolygon	Uses the specified <code>Brush</code> object to fill the interior of a polygon, which is represented by an array of points.
FillRectangle	Uses the specified <code>Brush</code> object to fill the interior of a rectangle.
FillRegion	Uses the specified <code>Brush</code> object to fill the interior of a <code>Region</code> object. We'll explore the <code>Region</code> class in Chapter 6 .

Note Most of the functions listed in [Table 2-2](#) make use of a `Pen` or `Brush` object. You use pens for drawing and brushes for filling and painting. We'll discuss pens and brushes in the [next chapter](#).

You may notice that some of the descriptions in [Table 2-2](#) are somewhat vague. This is because many of these methods are overloaded, which means that there are a number of different ways that you can use them to achieve the same thing. For example, where an operation requires a rectangle to be defined, the method is usually overloaded to accept a `Rectangle` object or a set of defining coordinates for the rectangle. You'll see how many of these methods operate as you progress through this book, but for a full reference listing, consult the GDI+ section of Microsoft's online MSDN documentation (<http://msdn.microsoft.com/library>).

Using Other Functionality of the Graphics Class

The `Graphics` class encapsulates more than just the functionality of drawing lines and filling shapes. Some of the additional functionality includes the following:

- **Clipping:** You can set a clipping region to be a `Rectangle`, a `GraphicsPath`, or a `Region`. It's useful to set a clipping region when you want to perform operations that affect only part of the drawing surface. After you've set a clipping area, the drawing operations you perform will *not* affect any pixels that lie outside the clipping area. We'll cover this in more detail in [Chapter 7](#).
- **Transformations:** You can use transformations to cause your drawing operations to be *scaled* (made larger or smaller), *translated* (shifted in the X and/or Y directions), or *rotated*. We'll look at transformations in [Chapter 8](#).
- **Font metrics:** You need to be able to determine the exact dimensions of given text when you supply a drawing surface and a font. When you draw a text string onto a drawing surface in a given font, the string will occupy a certain amount of horizontal and vertical space. This space is known as the *bounding rectangle* of the string. When you're drawing complex graphics composed of a number of elements, it's often useful to be able to express the bounding rectangle of text string elements drawn in a given font; for example, to control the exact relative positioning of these elements. We'll discuss text-drawing facilities in [Chapter 4](#).
- **Interoperability with GDI code:** Some GDI+ methods allow you to draw to a drawing surface using legacy code written using GDI in C++. Thus, if you have some C++ GDI code that performs some drawing operations, and you don't want to rewrite it in GDI+, you can create a window in the .NET Framework, then use the `Graphics.GetHdc` method to get a GDI Device Context (DC), and then

use the DC with the C++ GDI code to allow the C++ code to draw into your .NET window.

- **Interoperability with legacy applications:** There are also methods that allow you to use GDI+ functionality to draw to windows created in legacy applications. Thus, if you have a legacy application and wish to do some drawing using GDI+ (for example, because of the superior productivity of C#), you can create a window in the traditional way, then get a DC for the window, then use the `Graphics.FromHDC` method to create a `Graphics` object from the DC, and finally use GDI+ to draw into the window in your legacy application.

Now you know enough about drawing surfaces to be able to create one and start drawing some elements onto it. Next, you need to consider how to find your way around the drawing surface. Without a coordinate system, you wouldn't have a way to express the drawing operations you want to perform. In the [next section](#), we'll examine the coordinate system used by GDI+.

 Previous

Next 

 Previous

Next 

The GDI+ Coordinate System

When you're creating a complicated, intricate graphic, it is important that you draw exactly the pixels that you want to draw. When you're creating custom controls—which typically consist of rectangles, horizontal lines, and vertical lines—the resulting geometrical arrangements can make any misalignment particularly obvious to the human eye. In order to draw precision graphics successfully, you must understand the coordinate system of your drawing surface.

As discussed earlier in the chapter, the drawing surface itself is raster-based, so it is composed of pixels arranged in a two-dimensional grid. To reflect this, the GDI+ coordinate system consists of imaginary gridlines that run through the center of the pixels. The horizontal and vertical lines are numbered sequentially, starting at zero and incrementing in a downward or rightward direction. By default, the topmost and leftmost gridlines intersect at the pixel point (0, 0). [Figure 2-6](#) illustrates these imaginary gridlines. In the figure, the squares represent the pixels, and the gridlines are numbered along the top and down the left side of the grid.

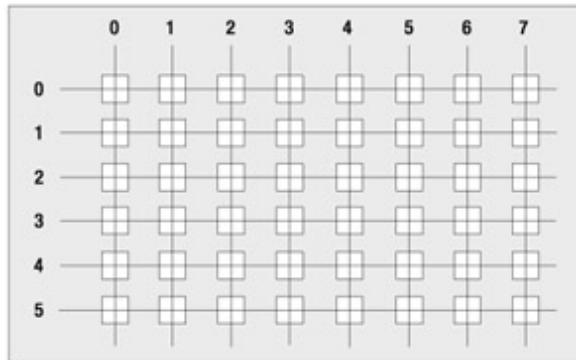


Figure 2-6: Coordinate gridlines

Drawing Lines

So what happens when you want to draw, say, a straight line? To draw a single line between two points, use the `Graphics.DrawLine` method (see [Table 2-2](#)):

```
e.Graphics.DrawLine(Pens.Black, 1, 2, 5, 2);
```

Here, we've specified the line as starting with pixel (1, 2) and ending at pixel (5, 2). The result of this operation is that five pixels have been drawn in black, as shown in [Figure 2-7](#).

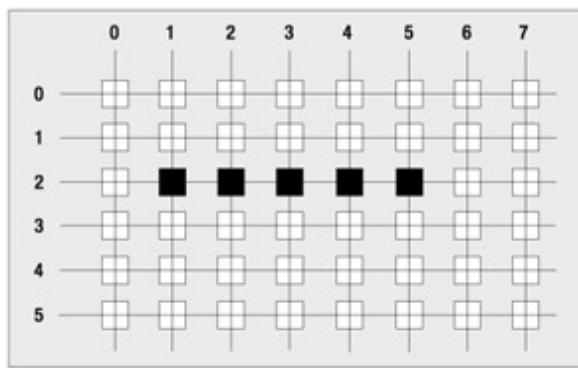


Figure 2-7: A horizontal line

It's worth noting how the length and width relate to the gridlines themselves. We've effectively asked for a vector line to be drawn between the points (1, 2) and (5, 2), and so we might have expected to see a line 4 pixels long, because the points (1, 2) and (5, 2) are exactly $5 - 1 = 4$ pixels apart. But as you can see from [Figure 2-7](#), the actual length of the line is 5 pixels, not 4 pixels. Why is that?

The difference is explained by the difference between a *point* and a *pixel*. A point is infinitely small. A pixel is not infinitely small; it is a square whose sides have a finite length. A line drawn between the points (1, 2) and (5, 2) is, in reality, drawn between the *center* of pixel (1, 2) and the *center* of pixel (5, 2), and does indeed have a length of 4 pixels. But here, GDI+ has decided to mark 5 pixels black. So we have a line whose length is effectively measured from the *left* of pixel (1, 2) to the *right* of pixel (5, 2), and that's a full 5 pixels long. Similarly (and perhaps more obviously), our line has a width of 1 pixel, not 0. The expression `Pens.Black` (which you'll meet formally in [Chapter 3](#)) represents a pen that draws lines that are black and 1 pixel wide. You'll see later in this section how GDI+ uses pen width and other criteria to decide which pixels to mark.

You draw a diagonal line from point (1, 1) to point (4, 4), with the following code:

```
e.Graphics.DrawLine(Pens.Black, 1, 1, 4, 4);
```

[Figure 2-8](#) shows the resulting drawing.

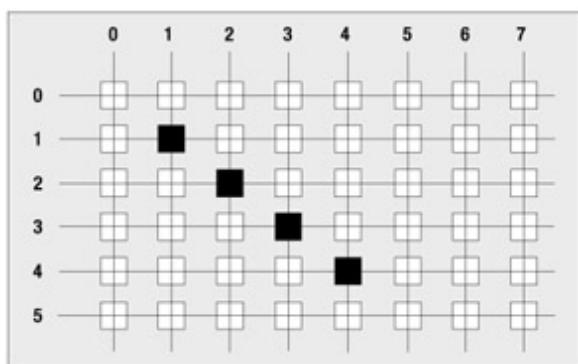


Figure 2-8: A diagonal line

Drawing Rectangles

The following code will draw a rectangle:

```
e.Graphics.DrawRectangle(Pens.Black, 1, 0, 5, 4);
```

The upper-left corner of this rectangle will be at the pixel centered at point (1, 0), as specified by the second and third arguments. The fourth and fifth arguments are the width and height of the rectangle, so the rectangle has a width of 5 pixels and a height of 4 pixels. The pen (`Pens.Black`) we use is defined by the first argument. The resulting rectangle looks like [Figure 2-9](#).

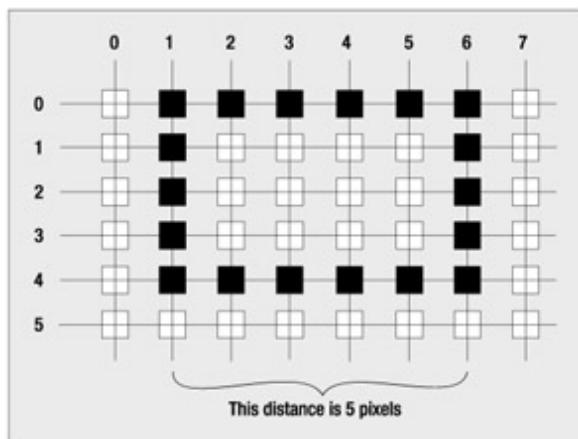


Figure 2-9: A rectangle with a width of 5 pixels and a height of 4 pixels

Again, the defining rectangle has a width of 5 pixels, but the thickness of the pen ensures that the actual width of the element drawn on the page "overhangs" by a half of a pixel on either side, so it's 6 pixels from end to end.

Similarly, the height of the rectangle appears to be 5 pixels, not 4 pixels, because the pen has a finite width (1 pixel), and is therefore not infinitely thin. It is this phenomenon—the width of the pen—that affects the length of the lines you draw. You'll learn more about the effects of pen width in [Chapter 3](#).

Filling a Rectangle

When you fill a rectangle, you actually fill in the entire area of the rectangle, not just draw the perimeter of the rectangle. To demonstrate, let's fill a rectangle that's defined using the same coordinates we used before, with the upper-left corner at point (1, 0), a width of 5 pixels, and a height of 4 pixels:

```
e.Graphics.FillRectangle(Brushes.Black, 1, 0, 5, 4);
```

The result is shown in [Figure 2-10](#). Given what you saw when you *drew* a rectangle using a pen width of 1 pixel, this isn't what you might have expected. As you can see, the width really *is* 5 pixels, not 6 pixels, and the height really *is* 4 pixels, not 5 pixels! Why is this?

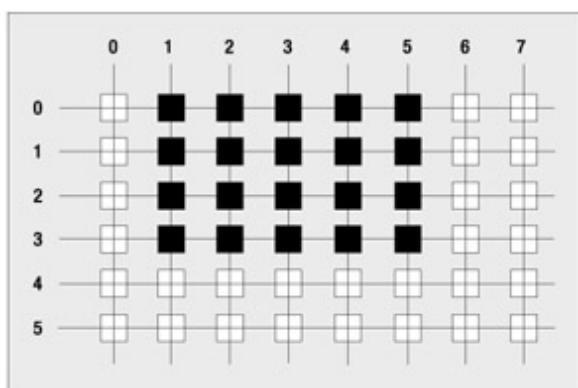


Figure 2-10: A filled rectangle

The reason comes in two parts. First, pen width is not relevant now. We're not drawing an *outline*; we're painting a *filled shape* (whose outline is infinitely thin). Second, given that pen width is not an issue, GDI+ knows that it is possible to supply a rectangle whose dimensions are exactly those specified (that is, 5 pixels by 4 pixels). But what it *can't* do is set the top-left corner of the filled rectangle into the center of the pixel (1, 0), because the smallest area it can work with is a *whole* pixel. Here, it must either *fill in* the pixel or leave it *unfilled*.

So, to satisfy the request, GDI+ fills in the arrangement of pixels shown in [Figure 2-10](#), consistently applying a rule that states that if the requested rectangle requires the bottom-right quadrant of a pixel to be filled, then the actual representation will fill the whole of that pixel. [Figure 2-11](#) illustrates how that rule deals with the rendering of the pixel (1, 1) in this example.

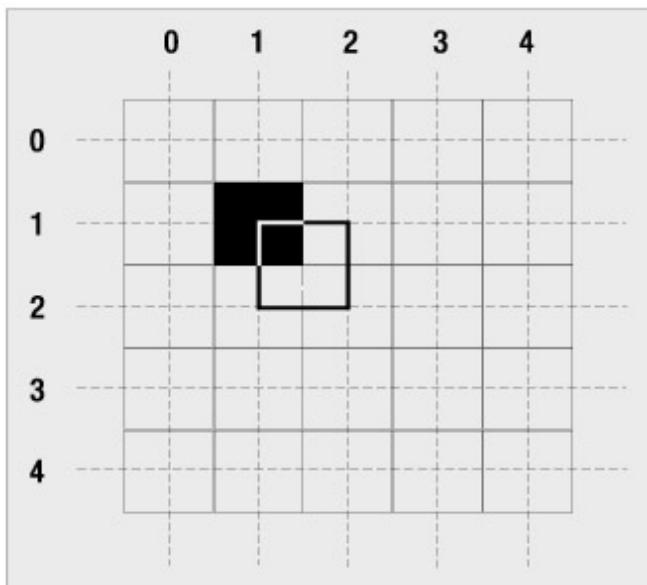


Figure 2-11: A very small rectangle and its pixel

But it would be better if the top-left corner of the rectangle really was set into the center of pixel (1, 0), or, at least, if it *appeared* to be that way. You can overcome this problem using an effect known as *anti-aliasing*.

Aliasing and Anti-Aliasing

Aliasing is the effect that you see as a result of the granularity of the pixels in your drawing surface. A classic example of this is when you draw a diagonal line. The pixel nature of the surface can only approximate the diagonal line, and so you see a staircase effect, as shown in [Figure 2-12](#).



Figure 2-12: An alias of a line

Aliasing is also an issue when you're drawing curved lines and text. It is called *aliasing* because the pixels on the screen are a kind of alias for the actual shape of your graphic.

Anti-aliasing is a technique that is used to get a better approximation of graphical operations like drawing curves, diagonal lines, and text. Anti-aliasing involves altering the colors of pixels near those that show the staircase effect. At a sufficiently high resolution, this gives the effect of smoothing, and thus counteracts the raster nature of the surface. [Figure 2-13](#) illustrates the effect. It is an anti-aliased representation of the diagonal line shown in [Figure 2-12](#).

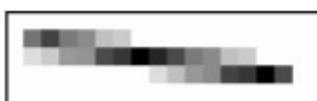


Figure 2-13: A line with anti-aliasing

Note that even after drawing with anti-aliasing, the resulting pixels are still an alias of the theoretical shape of the graphic. (In fact, the term *anti-aliasing* is a bit of a misnomer; the term *alias-reducing* might have been more appropriate.)

The purpose of anti-aliasing is to provide a better image for the user without a corresponding increase in hardware capabilities. With lower-resolution screens, anti-aliasing can have the effect of blurring the image somewhat. However, with high-resolution screens, the blurring effect is negligible, and the image quality is improved.

Let's see how anti-aliasing affects the rectangle we painted in the previous example, with its top-left

corner at point (1, 0), width of 5 pixels, and height of 4 pixels. We'll just add one extra line to the code, to turn anti-aliasing on:

```
e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
e.Graphics.FillRectangle(Brushes.Black, 1, 0, 5, 4);
```

Here, we're setting the `SmoothingMode` property of the `Graphics` object to the value `AntiAlias`. This value is part of the `SmoothingMode` enumeration, which belongs to the `System.Drawing` Drawing2D namespace. Therefore, for this code to work, you must remember to import the namespace into your code. [Figure 2-14](#) shows the result of this operation.

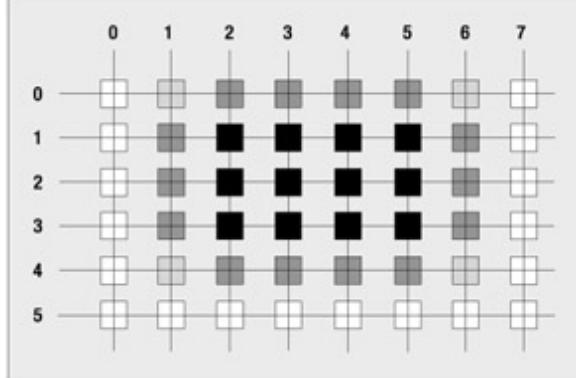


Figure 2-14: Filled rectangle with anti-aliasing

The color of the edge pixels is determined by their degree of intersection with the actual rectangle defined in the `FillRectangle` method call. So, the colors in this example are as follows:

- The light gray of the corner pixels reflects the fact that they have 25% intersection with the rectangle. It is a gray consisting of a 25% tint of black.
- The rectangle intersects 50% of each of the pixels along the sides of the rectangle, so the gray of those pixels consists of a 50% tint of black.

The actual drawn rectangle, when magnified, looks like [Figure 2-15](#). Of course, this would look much better when seen unmagnified!



Figure 2-15: Filled rectangle with anti-aliasing

Limitations of the Coordinate System

The GDI+ coordinate system and programming model are fine in situations where you're not concerned about *exactly* controlling each and every pixel within your custom control. If you are drawing wireframes, maps, and the like, anti-aliasing gives a nice appearance when viewed at a high resolution. However, this is not the most convenient coordinate system and programming model for writing custom controls, when you want to control the exact color of each and every pixel.

Some people might say that, given today's high-resolution screens, this is not really important. However, if two perpendicular lines that are supposed to meet at a corner don't *quite* meet, or if one line overshoots the corner by a single pixel, you can easily see this with the naked eye. Moreover, making controls with a

three-dimensional look requires a particularly high degree of attention to detail to get the appearance just right. A single pixel off in any direction can spoil the three-dimensional effect.

If you want to address certain pixels unambiguously, drawing in such a way that anti-aliasing is not used at all, you have two options:

- Turn off anti-aliasing, and then pass integer coordinates into all methods, paying close attention to the rules that GDI+ uses when drawing with anti-aliasing turned off.
- Pass floating-point coordinates into all methods, using coordinates always on the half pixel. With this option, it doesn't matter whether anti-aliasing is on or off; the result is the same.

Turning off anti-aliasing is not a good idea. It requires a thorough understanding of the seemingly arbitrary set of rules employed by GDI+ in an environment without anti-aliasing (in particular, the lower-right quadrant rule that you saw in the aliased rectangle shown in the [previous section](#)). This will result in code that is hard to read and maintain.

Using floating-point coordinates is a better option. For example, you can fill a rectangle with the upper-left corner at point (1.5f, 0.5f) with a width of 4 pixels and a height of 3 pixels, like this:

```
e.Graphics.FillRectangle(Brushes.Black, 1.5f, 0.5f, 4f, 3f);
```

The result will always look like [Figure 2-16](#), regardless of whether anti-aliasing is turned on or off.

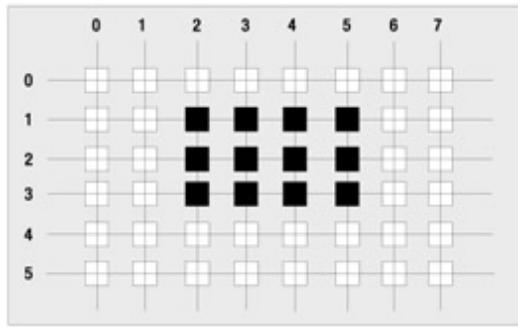


Figure 2-16: Rectangle drawn with floating-point coordinates

However, this technique also has definite drawbacks. In particular, the task of passing floating-point coordinates, always on the half pixel, will result in code that is harder to read, maintain, and explain to other developers.

In short, when designing coordinate systems for APIs, designers can take several different directions. The designers of GDI+ took a certain approach that is good for some types of applications, but is not so good for writing custom controls. So, in [Chapter 10](#), we will take a look at an alternative coordinate system and programming model that is more appropriate for writing custom controls. You'll see how to layer this alternative coordinate system over the existing GDI+ classes. This alternative coordinate system solves the problem of making readable, maintainable graphics code.

Units of Measurement

As you've seen, the default unit of measurement for specifying points on the drawing surface in GDI+ is the pixel. Unless you specify otherwise, the values of arguments that you send to methods such as `DrawRectangle` are assumed to be in terms of pixels. However, you can use other units of measurement, if necessary, as shown in [Table 2-3](#).

Table 2-3: GDI+ Units of Measure

Unit of Measure	Physical Size
Display	1/96 inch
Document	1/300 inch
Inch	1 inch
Millimeter	1 millimeter
Pixel	1 pixel (the default)

Point

1/72 inch

When writing custom controls, the most useful unit of measurement by far is the pixel. Your main concern when writing custom controls is that every pixel is drawn correctly. The problem you encounter when you try to use other units of measurement is that they describe points in the drawing surface that don't lie on a pixel boundary.

For example, if you're working with a display resolution of 96 DPI, and you use `DrawRectangle` to draw a rectangle of 0.1 inch per side, then the size of the resulting rectangle needs to be 9.6 pixels long. How would this rectangle be rendered on the screen? The only way that GDI+ can draw this is by an approximation, rounding up or down to the nearest pixel. For this reason, if you're working with other units of measurement, much more care is required when drawing custom controls to the screen.

Coordinate Data Types

You use quite a few data types with GDI+ to encapsulate coordinate information. Many of the GDI+ drawing methods take these data types as arguments. Some of these data types are structures, and some are classes. [Table 2-4](#) shows a summary of the GDI+ coordinate data types.

Table 2-4: GDI+ Coordinate Data Types

Data Type	Class/Structure	Description
Point	Structure	Represents a single point, expressed with integer precision. A <code>Point</code> structure contains <code>X</code> and <code>Y</code> properties.
PointF	Structure	Represents a single point, expressed with floating-point precision. A <code>PointF</code> structure contains <code>X</code> and <code>Y</code> properties.
Size	Structure	Represents a size, expressed with integer precision. A <code>Size</code> structure contains <code>Width</code> and <code>Height</code> properties.
SizeF	Structure	Represents a size, expressed with floating-point precision. A <code>SizeF</code> structure contains <code>Width</code> and <code>Height</code> properties.
Rectangle	Structure	Represents a rectangle, expressed with integer precision. Rectangles are expressed in terms of position and size (which may be expressed using coordinates and dimensions, or using <code>Point</code> and <code>Size</code> objects).
RectangleF	Structure	Represents a rectangle, expressed with floating-point precision.
GraphicsPath	Class	Encapsulates a series of connected lines, curves, arcs, pie shapes, polygons, rectangles, and more. We'll explore the <code>GraphicsPath</code> class in Chapter 6 .
Region	Class	Describes the interior of a graphics shape composed of rectangles and paths (<code>GraphicsPath</code> objects). We'll explore the <code>Region</code> class in Chapter 6 .

In this chapter, we will cover the basic data types of `Point`, `Size`, and `Rectangle` in more detail, including the utility methods and operator overloads that make it easy to work with these structures. The `GraphicsPath` and `Region` classes are quite a bit more complicated, and they are covered in [Chapter 6](#).

The Point Structure

You can construct a `Point` structure like this:

```
Point p = new Point(2, 1);
```

The `Point` structure offers a fair amount of functionality. [Table 2-5](#) lists the more commonly used methods and operators.

Table 2-5: Common Point Structure Methods and Operators

Method/Operator	Description
Offset(dx, dy)	Translates the Point by an offset dx horizontally and dy vertically
+ operator	Using the syntax pt1 = pt0 + sz, this translates the Point object pt0 by a horizontal and vertical distance given by a Size object
- operator	Like the + operator, but translates in the opposite direction
== operator	Compares two Point objects for equality, returning true or false
!= operator	Compares two Point objects for inequality, returning true or false
(Size)	A cast operator that converts the given Point object to a Size object

There is an explicit cast that converts from a Point to a PointF structure. All you need to do is to use the cast operator:

```
Point p = new Point(2, 4);
PointF pf = (PointF)p;
```

When converting from a PointF to a Point structure, you have three options:

- The Ceiling method rounds the values to the next highest integer.
- The Round method rounds to the nearest integer.
- The Truncate method truncates any digits to the right of the decimal point.

You can see how these three options work by running the following code:

```
PointF pf = new PointF(2.5f, 3.5f);
Point p1 = Point.Ceiling(pf);
Point p2 = Point.Round(pf);
Point p3 = Point.Truncate(pf);
Console.WriteLine("pf:" + pf);
Console.WriteLine("p1:" + p1);
Console.WriteLine("p2:" + p2);
Console.WriteLine("p3:" + p3);
```

The result is the following output:

```
pf:{X=2.5, Y=3.5}
p1:{X=3,Y=4}
p2:{X=2,Y=4}
p3:{X=2,Y=3}
```

The Size Structure

You can construct a Size object like this:

```
Size s = new Size(4, 3);
```

The Size structure provides the operators listed in [Table 2-6](#).

Table 2-6: Size Structure Operators

Operator	Description
+ operator	Using the syntax sz = sz0 + sz1, this returns a Size object whose height and width are the sums of the heights and widths of two other Size objects
- operator	Using the syntax sz = sz0 - sz1, this returns a Size object whose height and width are the differences between the heights and widths of two other Size objects
== operator	Compares two Size objects for equality, returning true or false
!= operator	Compares two Size objects for inequality, returning true or false

(Point)	A cast operator that converts the given Size object to a Point object
------------------	---

As with the **Point** class, when converting from a **SizeF** to a **Size** structure, you can use the **Ceiling**, **Round**, or **Truncate** method.

The Rectangle Structure

You can construct a **Rectangle** object from a **Point** object and a **Size** object, like this:

```
Point p = new Point(2, 1);
Size s = new Size(4, 3);
Rectangle r1 = new Rectangle(p, s);
```

Alternatively, you can construct a **Rectangle** object by expressing the **x** and **y** coordinates for the top-left corner and the dimensions (width and height) explicitly, like this:

```
Rectangle r2 = new Rectangle(2, 1, 4, 3);
```

The **Rectangle** structure has the methods and operators listed in [Table 2-7](#).

Table 2-7: Rectangle Structure Methods and Operators

Method/Operator	Description
Contains	Determines if the specified Point is contained in the rectangle
Inflate	Creates and returns an inflated copy of the specified rectangle
Intersect	Replaces the current rectangle with its intersection with a second, given rectangle
IntersectsWith	Determines if the current rectangle intersects with a given one
Offset	Translates the location of the rectangle
Union	Returns a rectangle that encloses the union of two specified rectangles
== operator	Compares two Rectangle objects for equality
!= operator	Compares two Rectangle objects for inequality

As with the **Point** and **Size** classes, when converting from a **RectangleF** to a **Rectangle** structure, you can use the **Ceiling**, **Round**, and **Truncate** methods.

Floating-Point Coordinates

We have already briefly discussed the capability of GDI+ to express coordinates in terms of floating-point numbers. If you are using floating-point coordinates, and if you have anti-aliasing turned on, GDI+ will use anti-aliasing to approximate noninteger coordinates. For example, if you consider a line that runs from point (2, 2) to point (20, 3.4), you would expect to see a line that has a slightly downward slope. We can draw this line with the following code (with antialiasing turned on):

```
Graphics g = e.Graphics;
g.FillRectangle(Brushes.White, this.ClientRectangle);
g.SmoothingMode = SmoothingMode.AntiAlias;
g.DrawLine(Pens.Black, 2f, 2f, 20f, 3.4f);
```

[Figure 2-17](#) shows the line on the screen at fairly high resolution (on the left) and also a close-up to see the individual pixels that make up the sloping line.



High-Resolution Display



Close-Up

Figure 2-17: Anti-aliased line

Admittedly, when you look at the close-up, it's hard to see a gently sloping line here! But these pixels, when viewed at a high resolution, *do* give the appearance of a slightly downward slope. This is because the pixels at the upper left are darker than the pixels at the lower left, and the pixels at the lower right are a little darker than the pixels at the upper right. You may have noticed that pixel (2, 2) is slightly lighter than pixel (3, 2). This is the correct behavior, in accordance with the anti-aliasing algorithms employed by GDI+.

There's another good reason to process coordinates in floating-point arithmetic. When performing a large number of transformations on a set of coordinates, if you were to keep the coordinates as integers, significant rounding errors would be introduced due to intermediate results being rounded to the nearest integer. You want to represent intermediate coordinates with high precision to preclude these types of errors.

The Coordinate System Origin

Each window or form into which you can draw has its coordinate space. If you create a custom control that can be put into other windows, this custom control has its own coordinate space. By default, the upper-left pixel of the custom control is (0, 0), regardless of where it is positioned within its parent window.

However, you can change the origin of the window by using transformations, which we will examine in detail in [Chapter 8](#).

Previous

Next

Previous

Next

Summary

Understanding the concept of a drawing surface is one of the key steps in mastering the GDI+ class library. Once you have an understanding of what a drawing surface is, what it represents, and how it is described, you can better understand how to use the drawing surface in combination with GDI+'s various methods and classes to create effective graphics. There are also a number of related issues that you need to understand. Here's a summary of the key points made in this chapter:

- A drawing surface is a raster-based representation of a form on a screen, a page being sent to a printer, or an image in memory. It's composed of a two-dimensional grid of pixels, and has three main characteristics: size, resolution and color depth. You represent your graphic by calling operations that are translated into a pixel-by-pixel representation, in which each pixel shows a particular (usually 24-bit) color.
- One of the interesting aspects of graphical class libraries like GDI+ is the juxtaposition between its vector-based programming interface, which describes the graphic, and the raster-based drawing surfaces on which the graphic is ultimately portrayed. By understanding how vectors are mapped and aliased to the drawing surface, you will be better able to write code that does exactly what you want it to do.
- The `GDI+ Graphics` class encapsulates the drawing surface abstraction, and you saw some examples that used methods of the `Graphics` class to write drawing code for any drawing surface.
- GDI+ represents colors predominantly using the RGB format, but also through a set of colors with predefined names and through the hue-based HSB model. The `Color` class represents a color. In GDI+, colors can contain an alpha component, which gives any color an aspect of transparency. While screens and printers use the alpha component only when drawing on the drawing surface, images can (optionally) retain alpha information, creating semitransparent images.
- GDI+ can apply anti-aliasing algorithms to your graphics, to eliminate the stairstep effect and thus create the appearance of smoother diagonal lines, curves, and text. You saw several examples of the use of this technique.
- The GDI+ coordinate system has advantages and disadvantages. ([Chapter 10](#) introduces an alternative coordinate system, which is intended as a solution to some of the problems of the GDI+ coordinate system.) You saw how you can express points and dimensions using the `Point`, `PointF`, `Size`, `.SizeF`, `Rectangle`, and `RectangleF` structures. We'll be using those

structures extensively throughout this book.

With all this information under your belt, you have a solid foundation for the [next chapter](#), in which we'll explore how to use pens to draw and brushes to paint.

 Previous

Next 

 PreviousNext 

Chapter 3: Pens and Brushes

Overview

When it comes to drawing graphics, *pens* and *brushes* are the basic tools of the trade. GDI+ has two classes that represent these two essential drawing tools. You use the `Pen` class to draw lines, curves, and outlines of shapes. You use the `Brush` class to fill shapes with colors and patterns. If you've read the first two chapters of this book, you've already seen the `Pen` and `Brush` classes used to demonstrate other concepts of GDI+.

These two classes are very flexible, giving you plenty of options that affect the way their operations are performed. By understanding these options, and the exact semantics of their methods, you'll be able to harness the full range and power they offer. This chapter covers the capabilities of the `Pen` and `Brush` classes, with many examples of their use.

First, we'll examine the `Pen` class. In particular, we'll look at pens with different widths, pens that draw dashed lines, pens that cap lines with arrowheads and other shapes, and the different ways you can join lines. We'll also look at the `Pens` class, which provides predefined pens for your use. Then we'll move on to look at the `Brush` class. We'll examine texture brushes, linear gradient brushes, and hatch brushes, as well as the predefined brushes in the `Brushes` class.

Next, you'll see how to use a brush to create a pen. This is a technique that allows you to use brush-type styles to perform pen-type operations, providing a great deal of extra flexibility in the types of effects you can achieve. Finally, I'll include a few words about how you can expect the instantiation of `Pen` and `Brush` objects to affect the performance of your applications.

 PreviousNext  PreviousNext 

Drawing with the Pen Object

You use the GDI+ `Pen` class to create custom pens. You can specify the `Color` and `Width` properties of the pen at the time you construct the `Pen` object. In fact, the `Pen` class is furnished with a number of other properties, such as the `StartCap` and `EndCap` properties, which allow you to add shapes to the start or end of lines, and the `DashStyle` property, which allows you to draw dashed and dotted lines. You can adjust the behavior of the `Pen` object any time after its instantiation by changing the values of these properties. We'll explore all of these capabilities in this section.

The Pen Class

The simplest pen is a solid pen with a width of 1 pixel. In this case, you are just required to specify the pen's color. So, to instantiate such a pen, you would use the class constructor like this:

```
Pen p = new Pen(Color.Black);
```

The following example creates such a pen, then uses it to draw a line from the pixel (0, 0) to the pixel (100, 100), and then calls the `Pen` object's `Dispose` method to release the resources used by the `Pen` object:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black);
    g.DrawLine(p, 0, 0, 100, 100);
```

[Figure 3-1](#) shows the resulting line.

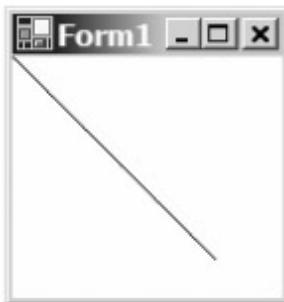


Figure 3-1: A diagonal line

Note The `FillRectangle` method here simply paints the background of the window (the client rectangle) white; as you can see, it uses a brush to do that. I've just included this operation in order to make the rendered examples look a little cleaner. We'll discuss brushes in detail later in this chapter.

In fact, there are four constructors for the pen class: two constructors allow you to specify a `Brush` object instead of a color, and two constructors give you the option of specifying the pen's width. The four constructors are as follows:

```
// Specified color, default width (1px)
Pen p1 = new Pen(myColor);
// Specified color, specified width (px)
Pen p2 = new Pen(myColor, myWidth);
// Specified brush, default width (1px)
Pen p3 = new Pen(myBrush);
// Specified brush, specified width (px)
Pen p4 = new Pen(myBrush, myWidth);
```

In these lines, `myColor` represents a `Color` object (specified in any of the ways discussed in [Chapter 2](#)), `myWidth` is a value of `Float` type, and `myBrush` is a `Brush` object. You'll see examples of all these constructors as we progress through this chapter. We'll consider using the constructors to combine pen and brush capabilities (by specifying a `Brush` object) toward the end of the chapter, after the discussion of brushes.

The GDI+ `Pen` class has a number of properties, offering you a large amount of control over how your `Pen` object draws graphic elements. Next, we'll take a look at some of the most commonly used properties of the `Pen` class.

The Width of the Pen

The first and most obvious thing to get right is the way you control a pen's width. The *width* of a pen in GDI+ means much the same as it does in real life: thin pens draw fine lines, and thick pens draw heavy lines. But there are a couple of additional subtleties that you need to consider.

To begin, let's draw a rectangle with a pen width of 1 pixel:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black);
    g.DrawRectangle(p, 3, 3, 8, 7);
    p.Dispose();
}
```

[Figure 3-2](#) shows the resulting rectangle. The top-left corner of this rectangle is the point with coordinates

(3, 3), and it is 8 pixels long and 7 pixels deep.

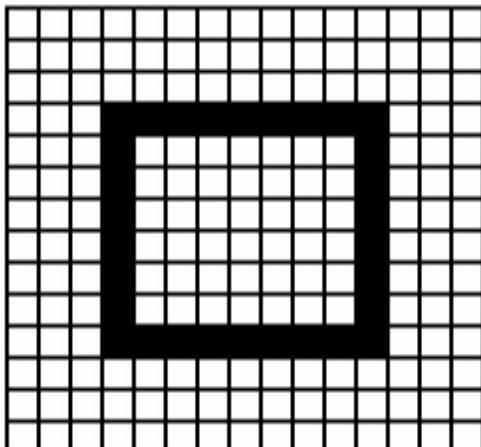


Figure 3-2: A 1-pixel-wide pen

Note In the illustrations here, the grid represents (part of) the client rectangle, and the top-left cell in the grid represents the pixel (0, 0).

As this example shows, the constructor that doesn't take a pen width as an argument creates a pen with a width of 1 pixel. Of course, you can create a pen with a width greater than 1 pixel. To do that, just use another one of the Pen class's constructors. Let's re-create the rectangle, this time using a pen width of 3 pixels:

```
Graphics g = e.Graphics;
Pen p = new Pen(Color.Black, 3);
p.Alignment = PenAlignment.Center;
g.DrawRectangle(p, 3, 3, 8, 7);
p.Dispose();
```

[Figure 3-3](#) shows the result of this code. Note two things about this rectangle. First, unsurprisingly, the sides of the rectangle are all 3 pixels wide. Second, although the top-left corner of the rectangle is defined to be the pixel (3, 3), you can see that the *apparent* corner of the rectangle is the pixel (2, 2). That's because the pen's width is greater than 1 pixel—it's 3 pixels—and so in this case, all the pixels adjacent to the pixel (3, 3) also are drawn black.

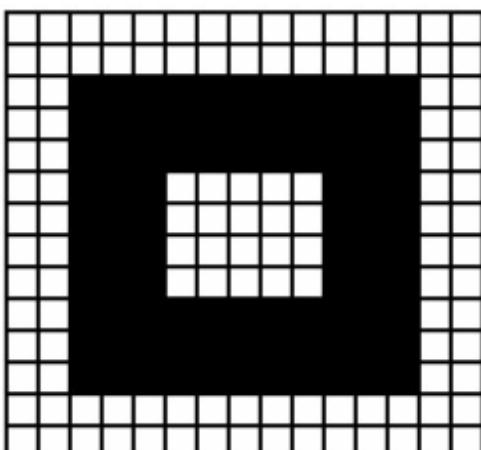


Figure 3-3: A 3-pixel-wide pen

This example uses the Pen object's Alignment property to control exactly how the pen should present its width on the drawing surface. It sets this property to Center, which dictates that the shape should be drawn as if the *center* of the pen described the shape. The best way to appreciate this is to take a look at what happens when you change the value of the Alignment property to Inset:

```
Graphics g = e.Graphics;
Pen p = new Pen(Color.Black, 3);
p.Alignment = PenAlignmentInset;
```

```

g.DrawRectangle(p, 3, 3, 8, 7);
p.Dispose();

```

Figure 3-4 shows the result. This time, the top-left corner of the rectangle, (3, 3), is the outermost, and the shape has been drawn as if the *outside* of the pen described the shape (that is, the pen ran along the *inside* of the shape).

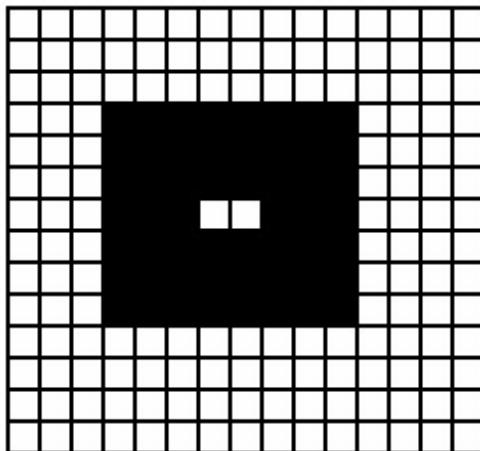


Figure 3-4: Line drawn with Inset alignment

The `PenAlignment` enumeration used here is part of the `System.Drawing.Drawing2D` namespace. This enumeration has three other values: `Outset`, `Left`, and `Right`. In my tests, I've found them to work in much the same way as `Center`, although Microsoft's documentation suggests that each has its own distinct effect (with `Outset` doing the opposite of `Inset`, and `Left` and `Right` creating a sort of left and right shadow, respectively).

Finally, note that if you don't set the `Alignment` property, it defaults to `Center`.

Dashed Lines

For some applications, you might want to draw dashed lines. For example, if you were creating a drawing program, you might want to use dashed lines to indicate that an object within the drawing space is selected. You have several different ways to create dashed lines and shapes. The simplest way is to set the `Pen` object's `DashStyle` property to one of the values provided by the `DashStyle` enumeration, like this:

```

private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Pen p = new Pen(Color.Black, 1);

    p.DashStyle = DashStyle.Dash;
    g.DrawLine(p, 3, 3, 100, 3);
    p.Dispose();
}

```

Figure 3-5 shows the dashed line produced by this code.

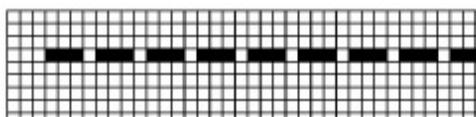


Figure 3-5: A dashed line

The `DashStyle` enumeration is provided by the `System.Drawing.Drawing2D` namespace, which offers five predefined styles: `Solid`, `Dash`, `DashDot`, `DashDotDot`, and `Dot`. Here is an example that uses all of these predefined styles:

```

p.DashStyle = DashStyle.Solid;
g.DrawLine(p, 3, 2, 100, 2);

p.DashStyle = DashStyle.Dash;
g.DrawLine(p, 3, 6, 100, 6);

p.DashStyle = DashStyle.DashDot;
g.DrawLine(p, 3, 10, 100, 10);

p.DashStyle = DashStyle.DashDotDot;
g.DrawLine(p, 3, 14, 100, 14);

p.DashStyle = DashStyle.Dot;
g.DrawLine(p, 3, 18, 100, 18);
p.Dispose();

```

[Figure 3-6](#) shows the line produced by each style.

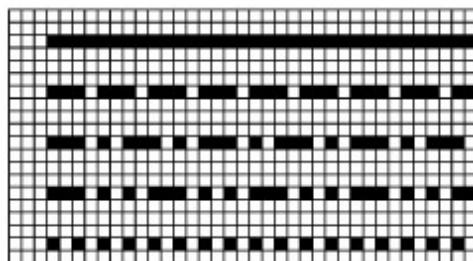


Figure 3-6: Various styles of dashed and dotted lines

Custom Dashed Lines

The predefined dash styles can be useful, but if you've tried viewing the result of the preceding code fragment at high resolution, you would probably find it difficult to tell them apart. For this reason, you might prefer to create your own customized dash styles. You can do this by using an array of integers that describe the pixel length of the dashes and the spaces between them. Here's an example:

```

private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Pen p = new Pen(Color.Black, 1);

    // The following line creates the custom dash pattern:
    float[] f = {15, 5, 10, 5};

    p.DashPattern = f;
    g.DrawRectangle(p, 10, 10, 80, 100);
    p.Dispose();
}

```

In this example, the dash pattern is specified by the array `{15, 5, 10, 5}`. It uses a four-element array, so the dash pattern will have a cycle of four: a 15-pixel dash, then a 5-pixel space, then a 10-pixel dash, and then a 5-pixel space. The pattern begins in the bottom-left corner, as shown in [Figure 3-7](#).



Figure 3-7: A custom dashed line

When you assign a value to the Pen object's DashPattern property, its DashStyle property is automatically set to the value DashStyle.Custom.

Wider Dashed Lines

If the pen's width is set to a value greater than 1 pixel, the actual length of each dash and space is calculated by multiplying the values specified in the array by the width of the pen. For example, let's try the previous custom dashed line with a pen width of 2 pixels:

```
Pen p = new Pen(Color.Black, 2);
```

```
// The following line creates the custom dash pattern:  
float[] f = {15, 5, 10, 5};
```

```
p.DashPattern = f;  
g.DrawRectangle(p, 10, 10, 80, 100);  
p.Dispose();
```

Now the pattern array {15, 5, 10, 5} produces alternating dashes of length 30 and 20, separated by spaces 10 pixels long. You can see this by comparing the result shown in [Figure 3-8](#) with the one shown in [Figure 3-7](#).

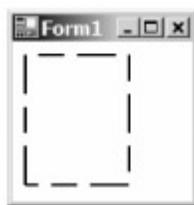


Figure 3-8: Wider custom dashed lines

This behavior means that the dashes and spaces in your lines are relative to the width of the pen. If you were creating a drawing program that allowed users to zoom in and out of their drawings, you would want this to happen. If you didn't have this behavior, as the user zoomed in, your nicely dashed line would not appear dashed in the same way.

Arrowheads and Other Line Caps

You can specify how GDI+ decorates the beginning and end of a line by using *line caps*. In practice, this means assigning values to the Pen object's StartCap and EndCap properties. The following code draws a line with a StartCap of LineCap.Round and an EndCap of LineCap.ArrowAnchor:

```
private void Form1_Paint(object sender,  
                         System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.SmoothingMode = SmoothingMode.AntiAlias;  
    g.FillRectangle(Brushes.White, this.ClientRectangle);  
  
    Pen p = new Pen(Color.Black, 10);  
    p.StartCap = LineCap.Round;  
    p.EndCap = LineCap.ArrowAnchor;  
    g.DrawLine(p, 30, 30, 80, 30);  
    p.Dispose();  
}
```

Note that you get the best results when you use a pen whose width is greater than 1 pixel. This example uses a 10-pixel width pen, which creates a very effective rounded end, as shown here:



[Table 3-1](#) shows the available line caps and their appearances.

Table 3-1: Available Line Caps

StartCap and EndCap Values	Result
ArrowAnchor	
DiamondAnchor	
Flat	
Round	
RoundAnchor	
Square	
SquareAnchor	
Triangle	

To create the capped lines, you use the `LineCap` values, as in the following example:

```
p.StartCap = LineCap.DiamondAnchor;
p.EndCap = LineCap.DiamondAnchor;
```

You can see that all of the anchor line caps (ArrowAnchor, DiamondAnchor, RoundAnchor, and SquareAnchor) have a very distinctive appearance, in that they add a decorative anchor to the end of the line. The others merely "trim" the line end in some way. All of these values belong to the `LineCap` enumeration, which is part of the `System.Drawing.Drawing2D` namespace.

If you were creating a diagramming program for a particular style of object-oriented analysis, you might want different end caps than those available by default. You can create custom line caps by instantiating the `CustomLineCap` class and setting the `CustomStartCap` and `CustomEndCap` properties of the `Pen` class. To create a custom line cap, you create a `GraphicsPath` object with the desired path of your line cap, then instantiate the `CustomLineCap` class using the `GraphicsPath` object (which is covered in [Chapter 6](#)). You can set various options regarding scaling, how lines join within the line cap, and so on. Building custom line caps is not necessary for most custom controls, so we will not delve into the subject too deeply, but it's useful to know that you can make your own line caps if you need them for your application.

Joined Lines

Suppose you were building an application that allowed users to draw lines representing electrical conduit. A conduit run is composed of straight lines and bends. Your GDI+ code would need to draw a number of straight lines that were joined where they met.

Whenever you perform a single operation that involves the rendering of joined lines, you can set the style in which the lines are joined. To do this, you use values from the `System.Drawing.Drawing2D` namespace's `LineJoin` enumeration, which has four values: `Miter` (the default), `Bevel`, `MiterClipped`, and `Round`. `MiterClipped` is not a common choice for writing custom controls, and we've been using `Miter` (by default) in the examples in this chapter. So here, let's just briefly demonstrate the other two.

The following code sets the `Pen` object's `LineJoin` property to the value `LineJoin.Bevel`, and then uses it to draw a rectangle:

```
private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Pen p = new Pen(Color.Black, 10);
    p.LineJoin = LineJoin.Bevel;
    e.Graphics.DrawRectangle(p, 20, 20, 60, 60);
```

Unsurprisingly, the resulting rectangle has beveled corners, like this:



Alternatively, you can set the `LineJoin` property to the value `LineJoin.Round`:

```
Pen p = new Pen(Color.Black, 10);
p.LineJoin = LineJoin.Round;
e.Graphics.DrawRectangle(p, 20, 20, 60, 60);
p.Dispose();
```

Then the rectangle has rounded corners, like this:



Once again, you can see that this feature produces better effects when the pen width is relatively high (but not so high that you can't make out the shape being drawn!). These images are produced with a 10-pixel pen width.

Predefined Pens

In [Chapter 2](#), you learned that the `Color` structure makes available 141 predefined colors. For convenience, GDI+ provides a predefined set of 141 colored pens. Each of these pens has a width of 1 pixel, and there is a prebuilt pen for every color defined in the `Color` structure.

To access these pens, use the `Pens` class, like this:

```
private void Form1_Paint (object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawRectangle(Pens.MistyRose, 20, 20, 40, 40);
}
```

The result of this code fragment is a rather pinkish-looking rectangle whose edges have a width of 1 pixel.

As a general rule, if your graphics allow you to use the pens from the `Pens` class, then you should do so. Because they are already built and ready to use, they offer some performance advantages. We'll return briefly to the subject of performance at the end of this chapter.

Note that when you create `Pen` objects using the `Pens` class, you don't create the pen explicitly, and therefore it's wrong to try to call the `Dispose` method of the resulting `Pen` object. If you attempted to do that, as in the following code fragment, then an exception would result.

```
Pen blackPen = Pens.Black;
g.DrawRectangle(blackPen, 20, 20, 40, 40);
blackPen.Dispose(); // generates an exception
```

In general, the rule to follow is this: if you created it, then you dispose of it. If you did not create it, then leave it alone. [Appendix C](#) discusses the dynamics of classes that implement the `IDisposable` interface.

Previous

Next

Previous

Next

Filling with the Brush Object

In GDI+, you use brushes to fill shapes with colors, patterns, and images. The GDI+ `Brush` class itself is an abstract class, and so you cannot instantiate it directly. Rather, the GDI+ API provides the five classes shown in [Table 3-2](#), which extend the `Brush` class and provide concrete implementations.

Table 3-2: GDI+ Brushes

Class	Description
<code>SolidBrush</code>	Fills a shape with solid colors
<code>TextureBrush</code>	Fills the shape with a raster-based image (BMP, JPG, and so on)
<code>LinearGradientBrush</code>	Fills the shape with a color gradient that changes evenly from one color to another, in a specified direction and between two specified (parallel) boundary lines
<code>PathGradientBrush</code>	Fills the shape with a gradient that changes evenly as you move from the boundary of the shape defined by the path to the center of the shape
<code>HatchBrush</code>	Fills the shape with various patterns

In this section, we'll look at some of the features of these brushes in more detail. It seems sensible to start with the simplest of these: the `SolidBrush`.

The SolidBrush Class

The following code creates a solid brush, and then uses it to fill a rectangle:

```
SolidBrush b = new SolidBrush(Color.Crimson);
g.FillRectangle(b, 20, 20, 40, 40);
b.Dispose();
```

This renders a rectangle whose interior is filled with crimson, as shown in [Figure 3-9](#).

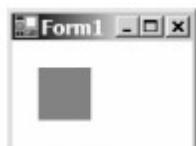


Figure 3-9: A filled rectangle

A `SolidBrush` has only one constructor, and after you instantiate it, no options can be set.

The TextureBrush Class

A `TextureBrush` object fills a shape with a raster-based image. It uses an image that comes from an image file such as a .bmp, .jpg, or .png file. You can obtain such an image from a file by using the `Bitmap` class, which is a subclass of the `Image` class (we'll discuss the `Image` class in detail in [Chapter 5](#)). To do that, use code like this:

```
Bitmap bmp;
bmp = new Bitmap("alphabet.gif");
```

In order to demonstrate a few simple examples that involve the `TextureBrush` class, we need a suitable image—one that has an obvious boundary and looks clearly different if we flip (or reflect) it horizontally or vertically. So, in these examples, we'll use the image `alphabet.gif`:



Note The image file `alphabet.gif` is included as part of the sample code for this book and available for download from the Downloads section of www.apress.com. For the example to work, you must place the image file in the same directory as the executable. If you're compiling with debugging information turned on, then Visual Studio .NET places the executable in the `/bin/debug` subdirectory of your project's directory, so you must place the image there.

For a first example here, let's just draw a simple rectangle, filled using this pattern:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("alphabet.gif");
    TextureBrush tb = new TextureBrush(bmp);

    g.FillRectangle(tb, 20, 20, 200, 70);
    bmp.Dispose();
    tb.Dispose();
}
```

Here, we first specify the name and location of our raster-based image as an argument to the `Bitmap` class constructor and specify the resulting `Bitmap` object in the constructor for the `TextureBrush` object. The result is a `TextureBrush` object that fills shapes using a tile effect based on the pattern in the image file.

Once you have your `TextureBrush` object, you can use it to create some filled shapes. In this example, we draw a filled rectangle, as shown in [Figure 3-10](#).

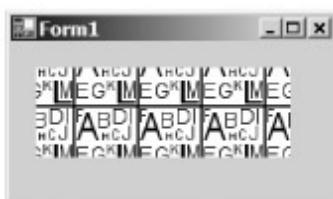


Figure 3-10: Rectangle filled with a TextureBrush

Notice that the top-left pixel of the rectangle painted here is (20, 20). You might be a little surprised to see only part of the `alphabet.gif` image nestling into the top-left corner of the rectangle. This is because the `TextureBrush` object fills the rectangle by using the image as a tile, arranging copies of the tile (image) on the drawing surface. Of course, it needs a starting point for the tiling pattern, and unless you tell it otherwise, it starts the tiling pattern from pixel (0, 0). With that in place, the top-left corner of the rectangle falls in the middle of a tile, and so you see only a part of the image in that corner.

Tip The `TextureBrush` behavior of tiling from the pixel (0, 0) is the default. You can override this default by setting a transformation on the `TextureBrush` object. See [Chapter 8](#) for details about transformations.

Overlapping Shapes

The way that the `TextureBrush` creates the fill allows you to draw a lot of shapes using the same brush, and if these drawing operations overlap, the bitmaps that make up the drawing operations will be aligned. To see how this works, let's add more overlapping shapes to the previous example. Modify the `Paint` event to draw another overlapping rectangle, as follows:

```
Graphics g = e.Graphics;
```

```

Bitmap bmp = new Bitmap("alphabet.gif");
TextureBrush tb = new TextureBrush(bmp);

g.FillRectangle(tb, 20, 20, 200, 70);
g.FillRectangle(tb, 45, 45, 70, 150);
bmp.Dispose();
tb.Dispose();

```

[Figure 3-11](#) shows how the overlapping shapes appear.



[Figure 3-11](#): Multiple fills using a TextureBrush

Selection from an Image as a Tile

You might have a large bitmap and want to use only a portion of it for your brush. The `TextureBrush` constructor is overloaded to allow you to select a portion of the image, which is then used as the tile when the `TextureBrush` fills. To demonstrate this, let's adjust the previous example to use the overloaded constructor, as follows:

```

Graphics g = e.Graphics;
Bitmap bmp = new Bitmap("alphabet.gif");
TextureBrush tb = new TextureBrush(bmp, new Rectangle(0, 0, 25, 25));

g.FillRectangle(tb, 20, 20, 200, 70);
g.FillRectangle(tb, 45, 45, 70, 150);
bmp.Dispose();
tb.Dispose();

```

As shown in [Figure 3-12](#), this example uses just a 25-pixel square taken from the top-left corner of `alphabet.gif` to tile the two rectangles.



[Figure 3-12](#): Using a selection from an image

Different Tiling Effects

You've seen the basic tiling effect that the `TextureBrush` achieves. You can use the `TextureBrush` object's `WrapMode` property to change the way it uses the tiles. Once again, GDI+ provides an enumeration, `WrapMode`, whose values can be used in conjunction with the `TextureBrush.WrapMode` property to produce different tiling effects, as you'll see in the following examples.

The `WrapMode` enumeration is part of the `System.Drawing.Drawing2D` namespace, and it contains five values: `Tile`, `Clamp`, `TileFlipX`, `TileFlipY`, and `TileFlipXY`. We'll briefly compare them

all here by using them, one at a time, to fill the entire client rectangle.

First, let's see the effect of setting the `WrapMode` property to `WrapMode.Tile`. This is the default, and it tiles the bitmap over the entire drawing surface.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("alphabet.gif");
    TextureBrush tb = new TextureBrush(bmp);

    tb.WrapMode = WrapMode.Tile;
    g.FillRectangle(tb, this.ClientRectangle);
    bmp.Dispose();
    tb.Dispose();
}
```

As you can see in [Figure 3-13](#), there isn't much new to note here, except that we're filling the *client rectangle*, instead of filling a smaller rectangle with specified corner pixels.

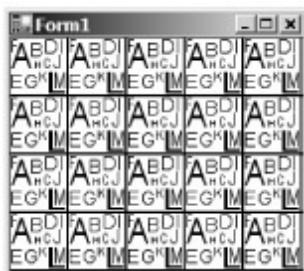


Figure 3-13: Using `WrapMode.Tile`

If you set the `WrapMode` property to `WrapMode.Clamp`, the texture brush doesn't tile the shape at all. Instead, it clamps the bitmap to the origin of the drawing surface.

```
tb.WrapMode = WrapMode.Clamp;
g.FillRectangle(tb, this.ClientRectangle);
```

[Figure 3-14](#) shows the results.



Figure 3-14: Using `WrapMode.Clamp`

If a clamped brush is used to fill a rectangle that is larger than the brush, any portion of the rectangle outside the image will not be drawn.

If you set the `WrapMode` property to `WrapMode.TileFlipX`, the texture brush tiles the bitmap over the drawing surface, flipping every other horizontal bitmap horizontally.

```
tb.WrapMode = WrapMode.TileFlipX;
g.FillRectangle(tb, this.ClientRectangle);
```

You can see this effect in [Figure 3-15](#).



Figure 3-15: Using WrapMode.TileFlipX

A similar effect is gained when you set the `WrapMode` property to `WrapMode.TileFlipY`.

```
tb.WrapMode = WrapMode.TileFlipY;
g.FillRectangle(tb, this.ClientRectangle);
```

As you can see in [Figure 3-16](#), this time, the texture brush flips every other *vertical* bitmap *vertically*.



Figure 3-16: Using WrapMode.TileFlipY

Finally, the effect of setting the `WrapMode` property to `WrapMode.TileFlipXY` gives an amalgam of the previous two effects, flipping the bitmap in both the vertical and horizontal directions.

```
tb.WrapMode = WrapMode.TileFlipXY;
g.FillRectangle(tb, this.ClientRectangle);
```

[Figure 3-17](#) shows the result.

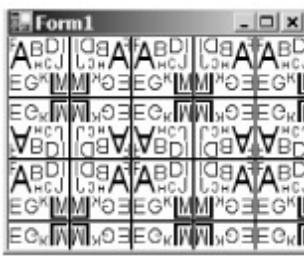


Figure 3-17: Using WrapMode.TileFlipXY

Image Attribute Specifications with TextureBrush

Before we move on to study the other `Brush` classes, it's worth mentioning that GDI+ allows you to affect how an image is drawn by using the `ImageAttributes` class to specify certain image attributes. We'll look at these techniques in more detail in [Chapter 5](#), but I mention it now because the `TextureBrush` can take advantage of these techniques.

GDI+ allows you to pass an `ImageAttributes` object to the `TextureBrush` constructor at the time you create the `TextureBrush` object. The `ImageAttributes` object can contain attribute values that affect the way the brush draws the image. To demonstrate this, here's a quick example that uses an `ImageAttributes` object to manipulate the version of `alphabet.gif` that is painted on the screen:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Bitmap bmp = new Bitmap("alphabet.gif");
    Graphics g = e.Graphics;
```

```

// Use a color matrix to change the color properties of the image
float[][] matrixItems = {
    new float[] {0.2f, 0, 0, 0, 0},
    new float[] {0, 0.8f, 0, 0, 0},
    new float[] {0, 0, 1, 0, 0},
    new float[] {0, 0, 0, 1, 0},
    new float[] {0, 0, 0, 0, 1}};
ColorMatrix colorMatrix = new ColorMatrix(matrixItems);

// Create an ImageAttributes object and set its color matrix
ImageAttributes imageAtt = new ImageAttributes();
imageAttSetColorMatrix(
    colorMatrix,
    ColorMatrixFlag.Default,
    ColorAdjustType.Bitmap);

// Create a TextureBrush object using the alphabet.gif image,
// adjusted by the attributes in the ImageAttributes object
TextureBrush tb = new TextureBrush(
    bmp,
    new Rectangle(0, 0, bmp.Width, bmp.Height),
    imageAtt);
tb.WrapMode = WrapMode.Tile;
g.FillRectangle(tb, this.ClientRectangle);
bmp.Dispose();
tb.Dispose();
}

```

The effect of all this is that the `TextureBrush` fills the client rectangle with a tile based on the image `alphabet.gif`, but whose background color is blue, not white. Unfortunately, the color doesn't come out too well in grayscale, but [Figure 3-18](#) gives you an idea of what to expect. This example makes use of objects and enumerations that belong to the `System.Drawing.Imaging` namespace. You'll learn more about color matrices and the `ImageAttributes` object in [Chapter 5](#).

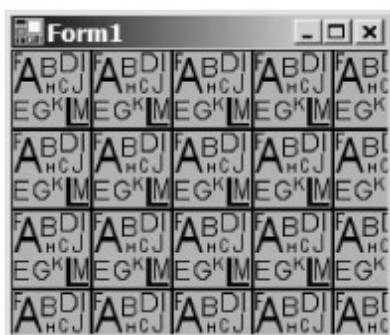


Figure 3-18: A TextureBrush that uses `ImageAttributes`

TILES AND ANTI-ALIASING

Texture brushes also behave in accordance with the coordinate system and anti-aliasing rules discussed in [Chapter 2](#). You may recall that the coordinate system model of GDI+ is such that coordinates refer to gridlines that run horizontally and vertically through the center of the pixels. What would it mean to draw an image at a location halfway through a pixel? If anti-aliasing is turned on, and you create a texture brush and then draw it on integral coordinates, the edges of the brush's bitmap will be anti-aliased. However, it is only the edges that are anti-aliased. The internal portion of the bitmap is not affected by the anti-aliasing option.

That said, drawing images with anti-aliasing is a dubious activity at best, and you will almost never want to do it. GDI+ will let you attempt it, but the results will probably not be what you want. It is far better to explicitly avoid doing anti-aliasing while drawing an image.

The LinearGradientBrush Class

A `LinearGradientBrush` object fills shapes with a linear *color gradient*. In its simplest terms, a color gradient consists of a gradual color change between two specified colors as you traverse a straight-line path that tends toward a specified angle. We'll demonstrate this with a couple of examples in a moment.

In fact, the `LinearGradientBrush` class encapsulates two-color gradients and multicolor gradients, as you'll see in this section. For a simple, *two-color linear gradient* brush, you can describe the gradient line either with a pair of points or with a rectangle and an angle. The `LinearGradientBrush` constructor is overloaded to take in these different possibilities, and you'll see them here.

A Gradient Described by Two Points

First, let's instantiate a `LinearGradientBrush` object by providing two points (a starting point and an ending point) and two colors (a starting color and an ending color):

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;

    LinearGradientBrush lgb = new LinearGradientBrush(
        new Point(0, 0),
        new Point(50, 10),
        Color.White,
        Color.Black);

    g.FillRectangle(lgb, this.ClientRectangle);
    lgb.Dispose();
}
```

[Figure 3-19](#) shows what you see when you run this application. Let's consider how this works. First, note that the two points specified describe the gradient line. Here, it starts at the pixel (0, 0) and ends at the point (50, 10). Of course, you can't see the gradient line in the figure, but you can imagine it. Second, note that at the start of the gradient line (the top-left corner of the client rectangle), the brush has painted the starting color, White. At the end of the line, the brush has painted the ending color, Black. In between, the color changes gradually from white to black as you look along this imaginary line.



Figure 3-19: Using two points to describe a gradient with `LinearGradientBrush`

Finally, note that this pattern is repeated in two directions—not horizontally and vertically, but in directions parallel and perpendicular to the direction of the gradient line. The effect is one of stripes that run perpendicular to the gradient line.

A Gradient Described by a Rectangle

Another way to define the gradient is to specify a rectangle and an angle in degrees. Here's an example:

```
LinearGradientBrush lgb = new LinearGradientBrush(
    new Rectangle(0, 0, 50, 50),
    Color.White,
    Color.Black,
    75f);

g.FillRectangle(lgb, this.ClientRectangle);
lgb.Dispose();
```

[Figure 3-20](#) shows the gradient produced by this example. In this case, the first color is used in the upper-left corner of the rectangle, and the second color is used in the lower-right corner of the rectangle. In between, the color changes as per the gradient specified by the fourth argument, which represents the angle that the gradient line tends to the horizontal.

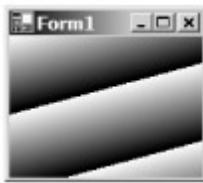


Figure 3-20: Using a rectangle to describe a gradient with `LinearGradientBrush`

Note that the gradient line is specified by the angle in the fourth argument, *not* by the positions of the two opposite corners of the rectangle! Note also that the angle specified represents the angle that the gradient line makes with the horizontal, in degrees. If you were to specify an angle of zero, the gradient line would be horizontal, and the stripes would appear to be vertical, as shown in [Figure 3-21](#).



Figure 3-21: A `LinearGradientBrush` with an angle of zero

Multicolor Gradients and Other `LinearGradientBrush` Options

At the beginning of the section, I mentioned that a `LinearGradientBrush` object can create both two-color and multicolor gradients. A *multicolor gradient* is like a sequence of top-to-tail two-color transitions, which pass through certain specified colors at certain points along the gradient line. You describe this sequence by specifying colors in the sequence and indicating how far along the gradient line you want each color to occur. For example, you might define a multicolor gradient to create a "chrome" look, like the effect you see when you look at a shiny bathroom faucet.

If you want to try a multicolor gradient fill, take a look at the `LinearGradientBrush` class's `SetInterpolationColors` method. It expects an array of the colors in the gradient, an array of numbers representing the positions, and an integer count of the total number of set colors in the sequence.

You can set other options when you're using gradient brushes. For example, you can control how one color transitions to another color. You can also enable and disable gamma correction. This is a complicated subject, but the essence of it is that you define a gradient between two colors, and with gamma correction, the gradient has the appearance of equal brightness at all points along the gradient.

These effects are quite advanced, and I won't go into detail about them here. With the increase in processing power, and the increase in users' expectations for a slick, fancy look, you can use gradients to good effect when building custom controls. But keep in mind that these effects can be overused. In general, you should stick to the norms of the Windows user interface. The Windows XP look contains some good examples of gradients that are subtly and tastefully implemented.

The `PathGradientBrush` Class

The `PathGradientBrush` object also provides a brush that fills a shape with a color gradient. In this case, you specify a path and two colors. The resulting color gradient begins at all points on the specified path and ends at a point that is deemed to be the center of the shape. This type of gradient reminds me of coloring maps in grade school, where we would color the outlines of the countries with a darker color than the center of the countries. You'll see an example in a moment.

In order to use the `PathGradientBrush` class effectively, you use it in conjunction with the

GraphicsPath class. So, let's take a quick look at the GraphicsPath class, which will be covered in detail in [Chapter 6](#), before we try a PathGradientBrush example.

You use the GraphicsPath class to describe a shape composed of multiple line segments. We say that a GraphicsPath object is *closed* if the path's ending point is the same as its starting point; otherwise, we say that the GraphicsPath object is *open*. You can convert an open GraphicsPath object to a closed one by connecting the last point to the first point. You can do this explicitly by calling the object's CloseFigure method. When you put together a GraphicsPath object, you first construct it, and then add segments to it. You can add different types of segments to the path. The following example creates a GraphicsPath object, adds three line segments to create an open path, then closes the path, and then draws the path with a pen.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsPath gp = new GraphicsPath();

    gp.AddLine(10, 10, 110, 15);
    gp.AddLine(110, 15, 100, 96);
    gp.AddLine(100, 96, 15, 110);
    gp.CloseFigure();

    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawPath(Pens.Black, gp);
    gp.Dispose();
}
```

The result looks like [Figure 3-22](#).



Figure 3-22: Line drawn with a GraphicsPath

Note In the GraphicsPath example, the second AddLine method call—`gp.AddLine(110, 15, 100, 96)`;—is not technically necessary. If you add two segments that are not joined, the GraphicsPath class will automatically add a segment to join them. In the example, for clarity, I showed how to add all segments explicitly, except for the last one, which is added by the call to CloseFigure.

This is not the whole story as far as the GraphicsPath class is concerned, but it's sufficient information to enable you to take a meaningful look at the PathGradientBrush class, as demonstrated in the following example:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsPath gp = new GraphicsPath();

    gp.AddLine(10, 10, 110, 15);
    gp.AddLine(110, 15, 100, 96);
    gp.AddLine(100, 96, 15, 110);
    gp.CloseFigure();

    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.SmoothingMode = SmoothingMode.AntiAlias;

    PathGradientBrush pgb = new PathGradientBrush(gp);
    pgb.CenterColor = Color.White;
```

```

pgb.SurroundColors = new Color[]
{
    Color.Blue
};

g.FillPath(pgb, gp);

g.DrawPath(Pens.Black, gp);
pgb.Dispose();
gp.Dispose();
}

```

In this example, we pass the `GraphicsPath` object to the constructor of the `PathGradientBrush` class. Then we need to set the `PathGradientBrush.CenterColor` property. Here, we've set it to white (using the `Color` class). The next statement is interesting:

```

pgb.SurroundColors = new Color[]
{
    Color.Blue
};

```

We pass an array of colors for the `SurroundColors` property. In this case, we created an array with only one color in it, so when you run the application, the `PathGradientBrush` paints blue at the edge of the path, transitioning to white at the center. This code paints the shape as shown in [Figure 3-23](#).

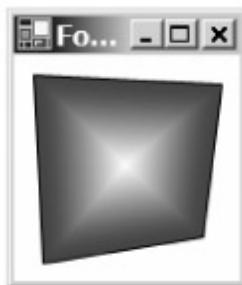


Figure 3-23: A `PathGradientBrush`

To see the full effect of the color array in the previous code fragment, let's modify our example to set the `SurroundColors` property to an array that contains several colors, like this:

```

pgb.SurroundColors = new Color[]
{
    Color.Blue,
    Color.Red,
    Color.Green,
    Color.Yellow
};

```

You'll need to try this out for yourself to appreciate the effect, because it doesn't show up too well in grayscale. You'll see a figure shaped similar to the one shown in [Figure 3-23](#), which is white in its center but whose colors vary around the edge of the graphic. The four corners (from top left, clockwise) are blue, red, green, and yellow.

Caution You cannot set the `SurroundColors` property to an array that contains more colors than there are segments in the `GraphicsPath` object. If you do so, you will get an exception.

The HatchBrush Class

The `HatchBrush` class provides a brush that can fill a shape with a variety of patterns. Hatch brushes are particularly useful when printing to a printer that is capable of printing only in black and white. For example, if you're drawing a bar chart or a map and filling adjacent regions with different colors, but the printer can print only in grayscale, adjacent regions might be difficult to differentiate. Instead, you can use hatch brushes, filling one region with a crosshatch, another region with diagonal lines, and so on, so that they are easily distinguishable.

The patterns for hatch brushes are always two-color patterns, based on a 16-pixel-by-16-pixel repeat.

This is a limitation within the .NET Framework of the hatch brush functionality. You set the hatch style via the `HatchStyle` enumeration, which belongs to the `System.Drawing.Drawing2D` namespace.

The following code sets the `HatchStyle` to the `Cross` pattern, and then fills the client area of the window:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;

    HatchBrush hb = new HatchBrush(
        HatchStyle.Cross,
        Color.White,
        Color.Black);
    g.FillRectangle(hb, this.ClientRectangle);
    hb.Dispose();
}
```

When run, the example looks like [Figure 3-24](#).

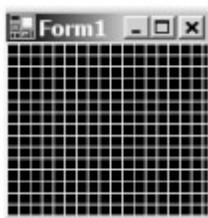


Figure 3-24: A HatchBrush with HatchStyle.Cross

You can set three options when creating a hatch brush: the foreground color, the background color, and the hatch style. There are 56 different hatch styles—too many to demonstrate here! For more information, see the `System.Drawing.Drawing2D.HatchStyle` enumeration in the .NET Framework documentation.

Predefined Brushes

GDI+ includes a `Brushes` class, which (like the `Pens` class discussed earlier) provides a set of predefined `SolidBrush` objects. There are 141 predefined brushes—one for each predefined color. In fact, you've already used this class several times in this chapter to fill the background of the client area of your window with white, like this:

```
g.FillRectangle(Brushes.White, this.ClientRectangle);
```

Each time you request a brush from the `Brushes` class, you do not *explicitly* create the brush, so you should not call the `Dispose` method on it. It seems that the intent of the Framework designers is that the Framework does not dispose of solid brushes acquired from the `Brushes` class until the application terminates. These brushes will typically be used often within an application, and the resource cost is minimal, so eliminating the creation/disposal cycle as far as possible provides better performance.

◀ Previous

Next ▶

◀ Previous

Next ▶

Creating a Pen from a Brush

Now that we've discussed the key features and differences of both pens and brushes, let's take a look at how you can use brush-type effects when you draw lines with a very wide pen. As an example, suppose that you have an image of water, and you want to create the effect that anywhere you draw a line, you let the image of water show through. Clearly, this will not be very interesting if your lines are only 1 pixel wide. Instead, using a wider pen, you can work as though you were filling the mark described by a pen with an effect that you can achieve only with a brush.

In the following example, we'll create a hatch brush, then create a pen from the brush, and then draw a rectangle with the pen. We'll create this effect with a hatch brush, but we could just as easily create the effect with a brush made from an image, or any other kind of brush. We'll give the pen a width of 8 pixels, which is wide enough to ensure that we can see the pattern of the hatch brush where the pen draws:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    HatchBrush hb = new HatchBrush(
        HatchStyle.WideUpwardDiagonal,
        Color.White,
        Color.Black);
    Pen hp = new Pen(hb, 8);
    g.DrawRectangle(hp, 15, 15, 70, 70);
    hb.Dispose();
    hp.Dispose();
}
```

To create the `Pen` object here, we've used one of the overloaded `Pen` constructors you saw near the beginning of this chapter. This constructor expects two arguments: the first argument is a `Brush` object and the second (optional) argument is an integer specifying the pen's width. When run, the example looks like [Figure 3-25](#).

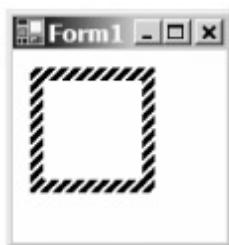


Figure 3-25: A Pen that uses a Brush

Previous

Next

Previous

Next

A Note about Performance

Most professional programmers have had exposure to pens and brushes in the context of graphical drawing. Some toolkits, such as GDI (the predecessor to GDI+), provide methods that combine the *shape-drawing* tasks of a pen and the *shape-filling* tasks of a brush into a single operation. However, as you've seen, in GDI+ these operations are completely separate.

If you want to draw an outline and fill the interior of it, you do it in two distinct operations. This is a very good thing. It results in drawing code that is cleaner and more straightforward. It doesn't really restrict you in any way. It just means that you must make two calls into the library if you want to both draw the outline of a shape and fill the shape.

As a developer, this separation of the drawing and filling tasks might cause you to be concerned about the potential effects on performance, particularly if each shape you draw requires two calls into the Windows API. As it turns out, the cost of marshaling an additional call into the Windows API is negligible, particularly when you compare this to operations such as changing pens and brushes. Experiments with the .NET Framework show a similar performance profile—changing pens is expensive compared with drawing operations with a pen.

If the separation of the drawing and filling tasks enables you to change pens and brushes less often, then the net result is a huge performance gain. Furthermore, if you organize your code so that you draw all lines of a given color before moving on to another color, your code executes much faster. The same is true of filling regions or drawing text.

A LESSON IN PROFILING AND OPTIMIZING CODE

When I needed to get a significant performance gain in some drawing code of a grid control, my first approach was to implement some sophisticated and complicated algorithms and data structures. These eliminated some work at drawing time, and I hoped they would give me the desired performance improvement. These optimizations helped, but not nearly as much as I wanted.

Frustrated, I spent some time profiling the various operations in the underlying toolkit. It was then that I realized the issue: the process of changing from one pen or brush to another is very slow. So, I made a very simple modification to the code: I arranged that the code first drew all of the cells in the grid that had a white background with black text (which was the most common case), and *then* drew all of the exceptional cases with different fonts, colors, and the like. I immediately got a performance increase of nearly an order of magnitude!

Aside from this specific lesson about changing pens and brushes in GDI+, there's a broader lesson that I took away from this experience: when profiling and optimizing code that makes extensive use of *any* underlying toolkit (not just graphical toolkits), be suspicious of the performance profiles of the various methods in the toolkit.

 Previous

Next 

 Previous

Next 

Summary

This chapter covered the essential details behind the fundamental GDI+ concepts of pens and brushes.

In GDI+, you generally use `Pen` objects for drawing shapes. You've seen that you can specify what color the pen should draw in, and you have the option of indicating what width the pen should be. You've also seen that the `Pens` class provides a set of 141 predefined colored pens, all with a 1-pixel width.

For filling operations, you use `Brush` objects. The `Brush` class itself is an abstract class, which manifests itself in the form of five special brush types: `SolidBrush`, `TextureBrush`, `LinearGradientBrush`, `PathGradientBrush`, and `HatchBrush`. We looked at how the `TextureBrush` uses an image file to create a textured effect, and how the `LinearGradientBrush` and `PathGradientBrush` use the color gradient technique to achieve their effects.

In passing, you saw that the `Pen` class is not the only class that allows you to draw shapes. In our study of the `PathGradientBrush`, we made use of the `GraphicsPath` class. You'll meet the `GraphicsPath` class again in [Chapter 6](#).

You also learned that you can perform single operations that combine the visual capabilities of a brush with the drawing capabilities of a pen. This gives you a technique that is just like drawing with a brush.

The basic set of functionality provided by `Pen` and `Brush` objects is enough in itself to enable you to draw powerful, effective graphics. But it gets even better when you combine these capabilities with other GDI+ functionality. In the [next chapter](#), you'll see how to use text and fonts to enhance your graphics even further.

 Previous

Next 

 PreviousNext 

Chapter 4: Text and Fonts

Overview

It's hard to think of an application that doesn't need to present text in some format or other. Some applications don't use much text, but most applications require at least some, even if it's just in dialog boxes. If you're building an elaborate custom control—such as a spreadsheet, a tree control, or network diagram—your use of text and fonts may be extensive. Therefore, if you're going to write custom controls, it's important for you to be able to write efficient and effective code for drawing text. To do this, you need to be able to identify exactly what you want your custom control to do, and you need a good understanding of the text-rendering capabilities of GDI+.

This chapter focuses on the text-rendering functionality provided by GDI+. As you read about the options, consider which ones will help you to achieve the requirements of your own applications. Here's what we'll cover in the chapter:

- An overview of font families, typefaces, fonts, and text
- How to specify coordinates when drawing text
- Text formatting (including horizontal and vertical centering, wrapping, vertical text, tab stops, and styles like strikethrough and underline)
- Drawing text using a GDI+ brush
- The metrics of fonts and text, including height and width
- How to control the quality of the text, focusing on anti-aliasing options
- Specifying font sizes in the page coordinate system

You've already seen one or two examples of text rendering in passing, in the earlier chapters of this book. This chapter serves not only to consolidate those examples, but also to provide a much more complete treatment of the issues surrounding the art of text rendering in GDI+.

Before we get into the various examples and practical points of this chapter, let's start by clarifying some basics about text and how GDI+ handles it.

 PreviousNext  PreviousNext 

An Overview of Text and Fonts

When we talk about a text string, we're really talking about a well-ordered string of characters, which is represented visually in some way in the drawing surface. A number of basic concepts are involved in this representation. So, let's begin by identifying those concepts and how they relate to one another. This is fairly straightforward, but it helps to set the scene for the rest of the chapter.

In order to store a text string, we need some system of characters that we can use to represent the string. The .NET Framework stores all text strings using the *Unicode* character-encoding system. Unicode maps every alphanumeric character, punctuation mark, and symbol (for every language) to a unique 16-bit (2 byte) number. (Note that the ASCII character representation scheme is 7-bit, which allows 128 characters, and the Extended ASCII scheme is 8-bit, which allows only a 256 character scheme.) Clearly, Unicode caters to a huge number of characters, and this is why a 16-bit representation is required. For this reason, some people refer to Unicode as *widetext*.

Note Prior to the advent of Unicode, there were hundreds of different encoding systems, including several different encoding systems for European character sets alone. One problem was that different systems used different codes to represent the same character. This meant that any text string would require a system-to-system translation, unless the applications involved were

using the same system. Another problem was that some of the systems that were designed to represent the characters of a particular language (such as English) didn't support a *complete* set of characters. So, having a Unicode-enabled application helps for porting to European languages. In fact, it's also essential for Far Eastern languages such as Japanese and Chinese, which have much larger character sets.

The character-encoding system itself helps us to distinguish the different characters in the character set, but it doesn't tell us what these characters look like. For that, we need glyphs and fonts. A *glyph* is a graphic symbol that provides the appearance or outline of a character. It is the graphical representation of an encoded character. (The word *glyph* is derived from the Greek word for *carving*.) A *font* is a mapping between a set of (Unicode) character codes and a set of glyphs. Thus, a font describes the appearance of each of the characters in the set. Thus, we describe a *text string* by a sequence of (Unicode) character codes, and we describe the *visual appearance* of that text string applying the font to each of the character codes.

Unicode characters are *not* glyphs. They provide only an abstract encoding of characters and symbols in a language. They have an *expected* appearance, but no *explicit* appearance. We require a font, which contains a description of the outline or physical representation of each character, to provide a set of glyphs that we can then paint onto our drawing surface.

Most of the fonts that we use in Windows today are *TrueType fonts*, which follow the model just described. The outlines of the glyphs of TrueType fonts are described mathematically, and therefore can scale to any size.

A few years ago, *raster fonts* (sometimes called *bitmap fonts*), were also popular. A raster font consisted of a raster-based representation of each character, for a few predetermined point sizes. If you wanted to use a raster font to draw a text string, but you wanted to draw it in a point size other than one supplied by the font, the rendering engine would attempt to interpolate a representation from the predefined point sizes, and the results were often quite unsightly.

In the past, the computations required to take a mathematical description of an outline of a glyph and render the glyph on a raster-based display were too intensive for the available processing power and therefore, for reasons of performance, we tended to use raster fonts. Today, however, our processors are more than powerful enough to handle such processing, so TrueType fonts are being used more commonly, and raster fonts will go the way of punch cards.

To complete this section, let's define four useful terms and their relationship to one another:

- A *glyph* is a description of the outline of a particular letter, numeral, or other symbol in a given font. The outline of the letter *E*, in 10-point Times Roman Bold, is an example of a glyph.
- A *font* is a mapping of a set of characters onto a set of glyphs, which hence describes what the characters should look like. An example of a font is 10-point Times Roman Bold.
- A *typeface* is a set of fonts, each of which looks the same but has a different size. Times Roman Bold is an example of a typeface.
- A *font family* is a group of typefaces that have a similar basic design, with certain variations in style. The variations in style from the normal are typically bold, italic, and bold-italic. Times Roman is an example of a font family.

Thus, a typeface is an instance of a font family with a particular style. A font is an instance of a typeface for a given size. A glyph is an instance of a font for a given character.

Note Often, in word processors, you will also see underline and strikethrough in the same dialog box where you specify the style of the font. However, you don't need separate typefaces to implement effects like underline and strikethrough. The rendering engine draws the underline or strikethrough after drawing the text, so a new typeface is not necessary.

With this understanding of the basic concepts involved in text rendering, we can now start to look at how it all works in GDI+.

Working with Fonts in GDI+

For working with fonts, GDI+ provides the `FontFamily` and `Font` classes. These classes belong to the `System.Drawing` namespace.

The `FontFamily` Class

In GDI+, you use an instance of the `FontFamily` class to represent a font family—a group of typefaces that have a similar basic design with certain variations in styles. As mentioned in the [previous section](#), the variations in style typically extend to bold, italic, and bold-italic only. For example, the Arial font family contains four typefaces:

- Arial Regular
- **Arial Bold**
- *Arial Italic*
- ***Arial Bold Italic***

You can construct a `FontFamily` object in a couple of different ways. For example, you can specify the name of the font family as a string, like this:

```
FontFamily ff = new FontFamily("Arial");
```

Alternatively, the `FontFamily` class provides a second constructor that expects a value from the `GenericFontFamilies` enumeration (which is part of the `System.Drawing.Text` namespace):

```
FontFamily ff = new FontFamily(GenericFontFamilies.Serif);
```

When you use this second overload of the constructor, you don't need to specify a font family by name. Rather, you tell the constructor what *type* of font family you want, and it then supplies one that meets your requirements. This enumeration lists three basic types of font, as shown in [Table 4-1](#). Of course, if you use this constructor, you lose some control over exactly which font is used.

Table 4-1: Font Family Types

Value	Description
Monospace	A monospace font family is one where all characters are the same width. Courier is an example of a monospace font family. This type of font is often used for program listings.
Serif	A <i>serif</i> is a small decorative line that is an embellishment or decoration added to the basic form of a character. Times Roman is a font family that has serifs.
SansSerif	A sans serif font family does not have serifs. Examples include Helvetica and Arial.

When designing professional controls, you may want to be more specific about the exact font family to use. In that case, the first constructor is a more appropriate choice. For example, if you were developing a grid control with text in cells, you may prefer to use a sans serif font such as Arial, as it presents a cleaner appearance to the user. However, if you were developing a custom control that presented a large amount of wrapping text to the user, you may prefer to use a font such as Times New Roman, because users are accustomed to reading large amounts of wrapping text in a serif font.

If you attempt to use a font that is not installed on the user's system, Windows will pick another font. If you want to make sure that a particular font is used, you should install that font with the application, and you need to verify that the font is installed before you use it. You can use the `InstalledFontCollection` object to do this, as you'll see in the "[Finding Available Fonts](#)" section later in this chapter.

The `Font` Class

Once you have created your `FontFamily` object, you can use it to define a specific `Font` object:

```
FontFamily ff = new FontFamily("Arial");
Font f = new Font(ff, 12);
```

Alternatively, you can define your font directly, by specifying the *name* of the font family in the first argument of the constructor:

```
Font f = new Font("Arial", 12);
```

You can use this constructor if you don't have (or don't want to create) a `FontFamily` object.

Note that if you need to know the sizes of detailed metrics for the font (such as the ascent and descent, which are the distance that the characters extend above and below the line, respectively), this functionality is available *only* via the `FontFamily` class. If you don't need access to these values, then the second constructor should do just fine. We'll return to the subject of font metrics in the "[Using Font Metrics for Precise Text Placement](#)" section later in this chapter.

Overloads for the Font Class Constructor

GDI+ provides quite a number of different overloaded constructors for the `Font` class. As you've seen, you can use a constructor that accepts a `FontFamily` object or the name of the font family as a string (though not both). In either case, you also need to pass in the desired size of your font.

Whichever of these options you choose, GDI+ also gives you the option to describe the required font further if you wish. You can identify the font style and the unit of measurement of the font. If you decide that you need to draw in italics, you can specify this at the time that you create the font. Moreover, another constructor allows you to create a `Font` object from another `Font` object, altering only the style.

Here's a full list of the `Font` class constructors offered:

```
// Create a font from another font
public Font(Font, FontStyle);

// Define the font via the FontFamily object
public Font(FontFamily, Size);
public Font(FontFamily, Size, FontStyle);
public Font(FontFamily, Size, GraphicsUnit);
public Font(FontFamily, Size, FontStyle, GraphicsUnit);
public Font(FontFamily, Size, FontStyle, GraphicsUnit, GdiCharSet);
public Font(FontFamily, Size, FontStyle, GraphicsUnit, GdiCharSet,
           VerticalFont);

// Define the font by passing the name of the font family in a string
public Font(FamilyName, Size);
public Font(FamilyName, Size, FontStyle);
public Font(FamilyName, Size, GraphicsUnit);
public Font(FamilyName, Size, FontStyle, GraphicsUnit);
public Font(FamilyName, Size, FontStyle, GraphicsUnit, GdiCharSet);
public Font(FamilyName, Size, FontStyle, GraphicsUnit, GdiCharSet,
           VerticalFont);
```

[Table 4-2](#) summarizes the arguments used in these `Font` constructors.

Table 4-2: Font Constructor Arguments

Argument	Description
Font	Allows you to easily create a new font that is a variation of an existing font.
Size	An integer that gives the em size of the constructed font, in terms of the <code>GraphicsUnit</code> . (A font's <i>em size</i> is a measurement that is proportional to its height.)
FontFamily	An existing <code>FontFamily</code> object (and hence specifies the font family of the desired font).
FamilyName	A string that contains the name of the font family.
FontStyle	Allows you to specify one of several styles for the font. Possible values for this argument are <code>Bold</code> , <code>Italic</code> , <code>Regular</code> , <code>Strikeout</code> , and <code>Underline</code> (these are the values of the <code>FontStyle</code> enumeration, which is part of the

	System.Drawing namespace). You can use the OR operator for this argument, to create a combination style, such as bold-italic.
GraphicsUnit	Specifies the unit of measurement used for the Size argument. The default GraphicsUnit for fonts is Point (1/72 inch). Other values are Display (1/96 inch), Document (1/300 inch), Inch, Millimeter, and Pixel (these are the values of the GraphicsUnit enumeration, which is part of the System.Drawing namespace).
GdiCharSet	A byte that specifies a GDI character set to use for this font. When using GDI (instead of GDI+), you can use one of a number of character sets, including ANSI_CHARSET, GREEK_CHARSET, and so on. The purpose of this option is backward-compatibility. If you are using only Unicode in your applications, you will not need to use this option.
VerticalFont	A Boolean argument that indicates whether the new Font object is derived from a GDI vertical font. A <i>vertical font</i> is one that allows you to draw vertical text (that is, text that lies along a vertical line).

Note Setting the GraphicsUnit for a font is different from setting the PageUnit for a Graphics object, although both are of type GraphicsUnit. The former controls the units with which you specify the size of the font; the latter controls the units of the position arguments of drawing operations.

As you can see, there are many Font constructors. However, it should be quite easy to pick the one that suits your requirements.

Font Disposal

The Font class, like many GDI+ classes, implements the IDisposable interface, which includes the Dispose method. It is important to call the Dispose method when you are completely finished with a Font object (or to wrap the use of the Font in a using block, so that it is disposed of at the end of the specified scope).

The usual rule applies: if you create it, then you must destroy it. For further review, read [Appendix C](#).

The Default Font

When you draw to a drawing surface on the screen (such as a Form object), a default Font object is available for your use. You don't need to create it or dispose of it; you get it from the Font property of the form. The following code demonstrates the use of this default Font object:

```
private void Form1_Paint(object sender,
                        System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Hello World", this.Font, Brushes.Black, 0, 0);
}
```

The DrawString method call expects a Font object to be specified in its second argument. Here, we've specified the Font property of the form itself, using the expression this.Font. The font is the default font for the window, and it will be the same font as that used for a button (unless you explicitly change the font for the button).

When building custom controls, there are three good reasons for using the Form object's Font property, rather than creating your own Font object, if possible:

- It is convenient. You don't need to create or dispose of this Font object.
- Text written in this font will match text for buttons, text boxes, radio buttons, and so on.
- When a localized application executes, the Font property will be set to a font that is appropriate to the local user interface culture.

 PreviousNext 

Drawing Basic Text

The easiest way to draw text onto a drawing surface is to use the `DrawString` method of the `Graphics` object, which draws the text to a specific location on the drawing surface. You've just seen an example of this in the "Hello World" example in the [previous section](#). In this case, we specified the location of the text by supplying X and Y coordinates, so that the text is placed at the position (0, 0) on the surface:

```
g.DrawString("Hello World", this.Font, Brushes.Black, 0, 0);
```

Drawing Text in a Bounding Rectangle

Another overload of the `DrawString` method takes a `RectangleF` structure as an argument and uses it as the bounding box for the text that you want to draw:

```
Graphics g = e.Graphics;
RectangleF drawRect = new RectangleF(150.0f, 150.0f, 200.0f, 50.0f);
g.DrawString("Hello World", this.Font, Brushes.Black, drawRect);
```

Note that when you define the height of this rectangle, you need to be sensitive to the `Height` property of the font (and in particular, the units used to express the font height and the rectangle height), so that the text will fit into the rectangle. In the example, the rectangle height is 200 pixels. The rectangle size is expressed in terms of the `PageUnit` of the `Graphics` class, and the `Font` object's `Height` property is given in terms of the current graphics unit. The default setting for both of these units is the pixel, and this makes it easy to compare the relative heights of the font and the rectangle. If you change one of these units (for example, by setting the `Graphics.PageUnit` property to `GraphicsUnit.Inch`), the comparison becomes much trickier.

Measuring Strings

An easier way to calculate both the height and width of a given text string rendered in a given font is to use the `MeasureString` method. This method returns a `Size` object, whose dimensions are given in terms of the current graphics unit. You'll see an example in a moment.

In fact, the pixel width of the string that you want to draw is a function of three parameters: the text string, the font, and the drawing surface. By default, when measuring the length of a string, GDI+ adds an extra length of 1/6 em. It does this to allow space for overhanging glyphs, such as the italic letter *f*. In addition, the width-calculating algorithm also adds space for something called *grid fitting*. This allows for the slight change in width that occurs when you take the mathematical description of the outline of a glyph and map it onto a raster-based drawing surface. (We'll discuss ems and other metrics for measuring fonts in detail later in this chapter, in the ["Using Font Metrics for Precise Text Placement"](#) section.)

Many times, when measuring strings, you don't want to have GDI+ add any extra length. In certain custom controls, you may be extremely particular about where text is drawn, and the exact metrics of the drawn text. In such a case, you don't want GDI+ to add the extra length; instead, you want to measure the text exactly.

You can instruct GDI+ to handle text in special ways using the `StringFormat` class. You can tell GDI+ to refrain from adding the additional space of 1/6 em by using a special `StringFormat` object, which you get by using the `StringFormat.GenericTypography` static property. Let's take a look at an example:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    FontFamily ff = new FontFamily("Times New Roman");
    Font f = new Font(ff, 12);
    String s = "Height: " + f.Height;
    SizeF sf = g.MeasureString(s, f, Int32.MaxValue,
        StringFormat.GenericTypographic);
    RectangleF r = new RectangleF(0, 0, sf.Width, f.Height);
```

```
g.DrawRectangle(Pens.Black, r.Left, r.Top, r.Width, r.Height);
g.DrawString(s, f, Brushes.Black, r, StringFormat.GenericTypographic);

f.Dispose();
}
```

This code obtains the height of the text to be displayed from the `Height` property of the `Font` object. It also obtains the height and width of the containing rectangle for the string, which is returned as a `SizeF` object. Then it creates and draws a rectangle whose width is given by the width of the bounding `SizeF` object (`sf.Width`) and whose height is given by the font's height, `f.Height`. Next, we use the `DrawString` method to draw the text string `s` using the font `f` within the bounding rectangle `r`.

When you run the example, you should see the result shown in [Figure 4-1](#). The rectangle is 19 pixels high, which means that, using the default coordinate system, GDI+ will draw 20 pixels in the vertical direction. The upper-left corner of the rectangle is located at coordinate (0, 0) on the drawing surface.



Figure 4-1: Font with a Height property of 19

Note Not all graphical toolkits specify location via the upper-left corner of the rectangle. Some, including GDI (though not GDI+), specify a point at the left end of the baseline instead. (The *baseline* of a font is the line that runs under the bottom of the majority of the characters.) Thus, GDI+ represents a deviation from GDI in this regard.

Drawing Text that Runs from Right to Left

The technique demonstrated in the [previous section](#) works well when you are drawing left to right. However, assembling many lines of text and making multiple calls to the `DrawString` method is not the best method for drawing strings from right to left, such as when you want to display text in Arabic, Hebrew, and other languages that read in right-to-left order.

The rules for drawing strings in languages that contain right-to-left formatting (or mixed right-to-left and left-to-right formatting) are defined by Unicode, and they are rather complicated. The good news is that you don't need to worry about these rules, because you can let GDI+ handle this whole issue automatically. If you're dealing with right-to-left formatted text strings, the best approach is to assemble the *entire* Unicode string, and then draw the string with a single call to the `DrawString` method.

Previous

Next

Previous

Next

Formatting Text

As noted in the description of measuring strings, the `StringFormat` class lets you specify special ways to handle text. Using this class, you can easily achieve many other common effects, such as the following:

- Change the appearance of the characters themselves (by making them normal, bold, italic, or bold-italic, or by adorning the characters with effects such as underlining or strikethrough).
- Draw longer strings of text and control the way that such strings wrap.
- Control positioning characteristics of the text; for example, by centering the text horizontally or vertically within a rectangle.
- Draw vertically oriented text (which is particularly useful for labels on tables, graphs, and bar charts).

- Use tabbing techniques to create columns of text.
- Create elaborate effects by using brushes to draw text.

In this section, you'll see how GDI+ makes these features available.

Setting Font Styles

The simplest way to format text, and probably the one that you will use most often, is simply to set the font style when you create the `Font` object. The five basic options for font style are `Bold`, `Italic`, `Regular`, `Strikeout`, and `Underline`. You can use a Boolean OR (or `|`) to combine these values together and get combinations of styles, such as bold-italic or bold-strikeout.

Here's a simple example that demonstrates this basic idea:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create some Font objects
    Font font      = new Font("Times New Roman", 12, FontStyle.Regular);
    Font bfont     = new Font("Times New Roman", 12, FontStyle.Bold);
    Font ifont     = new Font("Times New Roman", 12, FontStyle.Italic);
    Font bifont   = new Font("Times New Roman", 12,
                           FontStyle.Bold | FontStyle.Italic);
    Font sfont     = new Font("Times New Roman", 12, FontStyle.Strikeout);
    Font ufont     = new Font("Times New Roman", 12, FontStyle.Underline);
    Font bsfont   = new Font("Times New Roman", 12,
                           FontStyle.Bold | FontStyle.Strikeout);

    int h = font.Height;

    // Use each Font object to draw a line of text
    g.DrawString("Regular",           font, Brushes.Black, 0, 0);
    g.DrawString("Bold",             bfont, Brushes.Black, 0, h);
    g.DrawString("Italic",           ifont, Brushes.Black, 0, h*2);
    g.DrawString("Bold-Italic",       bifont, Brushes.Black, 0, h*3);
    g.DrawString("Strikeout",         sfont, Brushes.Black, 0, h*4);

    g.DrawString("Underline",          ufont, Brushes.Black, 0, h*5);
    g.DrawString("Bold & Strikeout", bsfont, Brushes.Black, 0, h*6);

    // Finally, dispose of all the Font objects
    font.Dispose();
    bfont.Dispose();
    ifont.Dispose();
    bifont.Dispose();
    sfont.Dispose();
    ufont.Dispose();
    bsfont.Dispose();
}
```

The example creates seven `Font` objects (each one representing the same font, but using a different style), and then uses each font to write one line of text. We've used the `FontStyle` enumeration (part of the `System.Drawing` namespace) to achieve this. When you run the example, you'll see the various styles, as shown in [Figure 4-2](#).



Figure 4-2: Various styles of a font

Drawing Multiline Text

In some other toolkits, the process of drawing multiline text is rather complicated. First, you need to calculate the pixel widths of different substrings to determine where to break the line between words, and then you draw each successive substring, placing it at exactly the correct location. In GDI+, none of this is necessary. You can easily draw multiline text using word wrapping. All you need to do is call the `DrawString` method, passing a rectangle whose width is less than the width of the text string and whose height is enough to house the multiple lines of text.

Here's an example of drawing multiline text:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Declare a long text string
    String s = "This string is long enough to wrap. ";
    s += "With a 200px-width rectangle, and a 12pt font, ";
    s += "it requires six lines to display the string in its entirety.";

    // Declare a Font object and a Rectangle object
    Font f = new Font("Arial", 12);
    Rectangle r = new Rectangle(20, 20, 200, f.Height*6);

    // Draw the rectangle.
    // Also draw the string, using the rectangle as a bounding box
    g.DrawRectangle(Pens.Black, r);
    g.DrawString(s, f, Brushes.Black, r);

    f.Dispose();
}
```

The result of this code looks like [Figure 4-3](#).

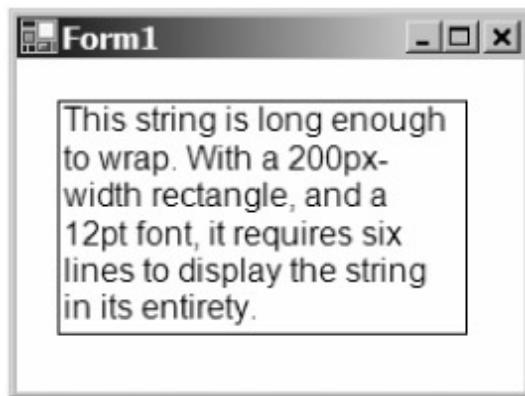


Figure 4-3: Wrapping text

Controlling the Number of Lines at Runtime

The previous example used a rectangle with a width of 200 pixels, and you could experiment to check that this rectangle was wide enough to contain the whole string in six lines. But what if, at design-time, you don't know the text string that is to be drawn? In that case, you certainly don't know the length of the string, and therefore you don't know whether the rectangle *r* will be big enough. The solution is to use the *MeasureString* method to calculate a *Size* object whose dimensions reflect the length of the string and any width restriction, and then use the *Size* object to set the width and height of the rectangle.

In the following example, we'll use this technique to draw some text into a rectangle that must be no more than 150 pixels wide, but can contain as many lines as necessary:

```
// Declare a long text string
String s = "This string is long enough to wrap. ";
s += "We'll use a 12pt font, and assume ";
s += "the text string must fit into a width of 150 pixels. ";

// Declare a Font object and a Rectangle object
Font f = new Font("Arial", 12);
SizeF sf = g.MeasureString(s, f, 150);
RectangleF rf = new RectangleF(20, 20, sf.Width, sf.Height);

// Draw the rectangle
// Also draw the string, using the rectangle as a bounding box
g.DrawRectangle(Pens.Black, rf.Left, rf.Top, rf.Width, rf.Height);
g.DrawString(s, f, Brushes.Black, rf);
```

In this example, GDI+ calculates at runtime that the string requires seven lines, as shown in [Figure 4-4](#).

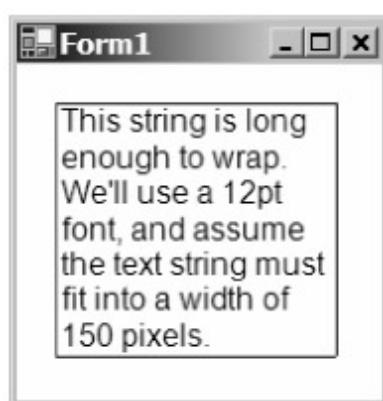


Figure 4-4: Wrapping text at runtime

Inserting Explicit Line Breaks

If you wish, you can explicitly control where a text string will break, by inserting newline characters (\n)

into the text in the appropriate places. For example, suppose we modified the above code as follows:

```
String s = "This string is long enough\n to wrap. ";
```

Now the code will force a line break after the word *enough*.

Preventing Text Wrapping

You can also *prevent* wrapping of text by creating a `StringFormat` object, setting the `FormatFlags` property to `StringFormatFlags.NoWrap`, and passing the `StringFormat` object to the `DrawString` method. Here's an example:

```
// Declare a long text string
String s = "This string is long enough to wrap. ";

// Declare a Font object and a Rectangle object
Font f = new Font("Arial", 12);
Rectangle r = new Rectangle(20, 20, 150, f.Height*4);

// Declare the StringFormat object, and set the FormatFlags property
StringFormat sf = new StringFormat();
sf.FormatFlags = StringFormatFlags.NoWrap;

// Draw the rectangle
// Also draw the string, using the rectangle as a bounding box
g.DrawRectangle(Pens.Black, r);
g.DrawString(s, f, Brushes.Black, r, sf);
f.Dispose();
```

The value `StringFormatFlags.NoWrap` belongs to the `StringFormatFlags` enumeration, which is part of the `System.Drawing` namespace. When you run this code, the text doesn't wrap but instead disappears under the right edge of the rectangle, as shown in [Figure 4-5](#).

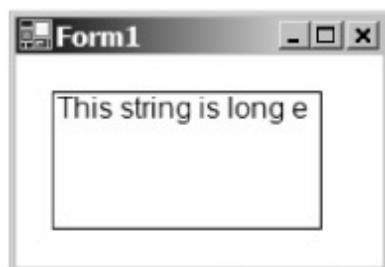


Figure 4-5: Text drawn with `NoWrap`

Centering Text Horizontally and Vertically

In the previous examples, the text is (by default) *horizontally aligned* to be flush with the left edge of the containing rectangle. But, of course, you may not always want this alignment. You may want the text to be center-aligned or flush with the right edge.

In fact, in those examples, the text is also (by default) *vertically aligned* to be flush with the top edge. This isn't so obvious, because in all of those examples, the containing rectangle is just large enough to completely contain the text. But given a bit more space in your rectangle, you may also want to align the text vertically so that it is centered or close to the bottom edge.

Setting the Text Alignment

To set the text alignment, you can once again take advantage of the features of the `StringFormat` object. This time, set its `Alignment` and `LineAlignment` properties to the appropriate values, and then pass it to the `DrawString` method.

In the following example, we'll place the text in a big rectangle with a lot of extra white space and center-align the text horizontally and vertically.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
```

```
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, this.ClientRectangle);  
  
    // Create a text string, a Font object, and a StringFormat object  
    String s = "This is a long string that will wrap. ";  
    s += "It will be centered both vertically and horizontally.";  
    Font f = new Font("Arial", 12);  
    StringFormat sf = new StringFormat();  
  
    // Center each line of text horizontally and vertically  
    sf.Alignment = StringAlignment.Center;           // horizontal alignment  
    sf.LineAlignment = StringAlignment.Center;        // vertical alignment  
  
    // Draw a rectangle and the text string  
    Rectangle r = new Rectangle(10, 10, 300, f.Height*4);  
    g.DrawRectangle(Pens.Black, r);  
    g.DrawString(s, f, Brushes.Black, r, sf);  
    f.Dispose();  
}
```

The text is formatted as shown in [Figure 4-6](#).

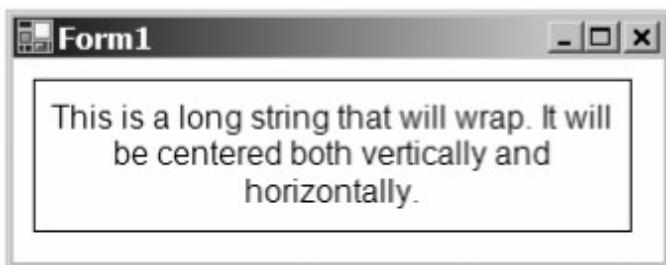


Figure 4-6: Centered text

Drawing Vertical Text

You can draw *vertically oriented text* in two ways. One way is to define some horizontally oriented text (such as the text in any of the previous examples), and then transform the coordinate space with a rotation through 90 degrees. We won't cover this technique here; we will examine it in [Chapter 8](#), along with the other transformation techniques.

A second, and much easier, way to draw vertical text is to create a `StringFormat` object *specifically* to draw vertical text. One obvious advantage of this technique over the transformation technique is that you don't change the orientation of the entire coordinate system! Of course, sometimes you want to change the orientation of the coordinate system. For example, if you want to draw your entire document rotated 90 degrees to the right, then using the `StringFormat` object to rotate text is not appropriate.

To draw vertical text, use an overload of the `StringFormat` class constructor, passing the value `DirectionVertical` (from the `StringFormatFlags` enumeration, which belongs to the `System.Drawing` namespace). The following code demonstrates vertical text drawing in this manner:

```
private void Form1_Paint(object sender,  
                         System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, this.ClientRectangle);  
  
    // Create a text string, a Font object, and a StringFormat object  
    String s = "Accrington Stanley";  
    StringFormat sf = new StringFormat(StringFormatFlags.DirectionVertical);  
    Font f = new Font("Times New Roman", 14);  
  
    // Calculate the size of the text string's containing rectangle  
    SizeF sizef = g.MeasureString(s, f, Int32.MaxValue, sf);
```

```
// Create and draw the rectangle
// Also draw the text string (using the StringFormat object)
RectangleF rf = new RectangleF(20, 20, sizef.Width, sizef.Height);
g.DrawRectangle(Pens.Black, rf.Left, rf.Top, rf.Width, rf.Height);
g.DrawString(s, f, Brushes.Black, rf, sf);

f.Dispose();
}
```

In this code, we call a version of the `MeasureString` method that expects a `StringFormat` object passed as the fourth argument. The dimensions of the resulting `Size` object take into account the fact that we plan to orient the text vertically. In this case, the `Size` object is much taller than it is wide. Thus, we can use those dimensions—`sizef.Width` and `sizef.Height`—explicitly to set the width and height of the rectangle in which the text string is drawn. [Figure 4-7](#) shows the vertical text that results.



Figure 4-7: Vertically drawn text

While this technique for drawing vertical text is easier than applying a rotation transformation, it's also less flexible. If you use a rotation transformation, you have the freedom to draw the text at *any* angle. The technique shown here allows you to orient your text vertically only downwards (at a 90-degree clockwise rotation).

Setting Tab Stops

One particularly nice feature of GDI+'s text string formatting system is the way it implements *tab stops*. If you want to use tab stops, you first need to set them. You do this using the `StringFormat` object's `SetTabStops` method. Once you've set your tab stops, you can use the tab stops in the text that you want to draw by including the tab character (`\t`) in the text string itself, and passing the text string and the `StringFormat` object to the `Graphics.DrawString` method.

In the following example, we'll use a set of four tab stops to create the effect of a table with four columns.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a couple of Font objects
    Font f = new Font("Times New Roman", 12);
    Font bf = new Font(f, FontStyle.Bold);

    // Create a StringFormat object, and set the tab stops, in pixels
    StringFormat sf = new StringFormat();
    float[] ts = {10.0f, 70.0f, 100.0f, 90.0f };
    sf.SetTabStops(0.0f, ts);
```

```

// Create some text strings
// The \t escape-sequence in these lines specifies the tab
string s1 = "\tName\tHair Color\tEye Color\tHeight";
string s2 = "\tBob\tBrown\tBrown\t175cm";

// Use the text string, Font, StringFormat, etc.
// to draw the text strings
g.DrawString(s1, bf, Brushes.Black, 20, 20, sf);
g.DrawString(s2, f, Brushes.Blue, 20, 20 + bf.Height, sf);

f.Dispose();
bf.Dispose();
}

```

In the first few highlighted lines here, we create a `StringFormat` object and an array of four float values, and we assign the array to the `StringFormat` object's `SetTabStops` property. The array represents the number of tab stops that we want to set and the width of each. So, the first tab stop will be 10 pixels from the left of the containing rectangle; the second will be 70 pixels further along; the third will be 100 pixels after that, and the fourth is 90 pixels after that. Hence, the position of the first tab is relative to the left edge of the rectangle, and every subsequent tab stop is positioned relative to the previous tab.

Note In this example, the tab stop widths specified will be measured in pixels. GDI+ uses whatever unit is set for the `PageUnit` property of the `Graphics` object. In this case, it's the default, which is the pixel. You'll meet the `PageUnit` property later in this chapter, in the "[Drawing Device-Independent Graphics](#)" section.

With the tab stop positions set, all you need to do is use them. To do that, create some lines of text that include `\t` escape sequences and draw them (using the `Graphics.DrawString` method and specifying your `StringFormat` object). The result is as shown in [Figure 4-8](#).



Figure 4-8: Tabbed text

In this example, our first tab is a 10-pixel tab, and we've placed a `\t` at the start of each text string. Its effect is to create the appearance of an empty 10-pixel column at the left end of the table.

Mixing `\t` and `\n`

You can also put newline characters (`\n`) into the text, and hence draw multiple lines of text that are separated by tab stops. Here's an example:

```

// Create some text strings
// The \t escape-sequence in these lines specifies the tab
string s1 = "\tName\tHair Color\tEye Color\tHeight";
string s2 = "\tBob\tBrown\tBrown\t175cm";
string s3 =
    "\tMary\tBlond\tHazel\t161cm\n\tBill\tBlack\tBlue\t168cm";

// Use the text string, Font, StringFormat, etc.
// to draw the text strings
g.DrawString(s1, bf, Brushes.Black, 20, 20, sf);
g.DrawString(s2, f, Brushes.Blue, 20, 20 + bf.Height, sf);
g.DrawString(s3, f, Brushes.Blue, 20,
                20 + bf.Height + f.Height, sf);

```

This gives the results shown in [Figure 4-9](#).

The screenshot shows a Windows application window titled "Form1". Inside the window, there is a table with four columns: "Name", "Hair Color", "Eye Color", and "Height". The data is as follows:

Name	Hair Color	Eye Color	Height
Bob	Brown	Brown	175cm
Mary	Blond	Hazel	161cm
Bill	Black	Blue	168cm

Figure 4-9: Multiline tabbed text

Simulating Right-Justified Tab Stops

GDI+ does not have any facilities for creating centered or right-justified tab stops. However, if you use a fixed-width (monospace) font, you can simulate right-justified tabs by using the `String.Format` method to right-justify text within the string that you draw using the `DrawString` method. In this example, we do just that:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Font f = new Font("Courier New", 12);
    Font bf = new Font(f, FontStyle.Bold);

    // Create a StringFormat object, and set the tab stops, in pixels
    StringFormat sf = new StringFormat();
    float[] ts = { 100.0f, 100.0f, 90.0f };
    sf.SetTabStops(0.0f, ts);

    // Create a string that describes how the specified values
    // should be formatted
    string fs = "{0}\t{1,8}\t{2,8}\t{3,8}";

    // Create some formatted text strings
    string s1 = String.Format(fs, "Month", "Revenue", "Expense", "Profit");
    string s2 = String.Format(fs, "January",
        "900.00", "1050.00", "-150.00");
    string s3 = String.Format(fs, "February",
        "1100.00", "990.00", "110.00");

    // Draw the text strings
    g.DrawString(s1, bf, Brushes.Black, 20, 20, sf);
    g.DrawString(s2, f, Brushes.Blue, 20, 20 + bf.Height, sf);
    g.DrawString(s3, f, Brushes.Blue, 20, 20 + bf.Height + f.Height, sf);

    f.Dispose();
    bf.Dispose();
}
```

The monospace font we've chosen here is Courier New. We also set three tab stops (of widths 100, 100, and 90 pixels). Then we create a "template" formatting string, `fs`, which describes how each of the strings is to be formatted. The format string has the value `{0}\t{1,8}\t{2,8}\t{3,8}`. This looks complicated, but it can be interpreted easily. Each of the expressions in {} brackets represents a value. The \t characters indicate where we use the tab stops. So, each text string that we draw will consist of four values separated by three tab stops.

So, what are the values? We specify them in the strings that we define in the subsequent lines, using the `String.Format` method. This method expects a formatting string and a list of values. The values are indexed using a zero-based index (0, 1, 2, 3). When we draw the string, we place the zeroth value in place of `{0}`, the first value in place of `{1,8}`, and so on, to get something like this:

The second value in the {} bracket expressions, if it exists, is an integer that indicates whether the value should be left or right-justified and the character width of the value. The value will be padded out with spaces if it's too small. If the value is positive, it will be right-justified. If the value is negative or absent, it will be left-justified. In this case, we get one left-justified column and three right-justified columns that are eight characters in width:

```
Month      \t Revenue\t Expense\t Profit
```

The result is as shown in [Figure 4-10](#).

Month	Revenue	Expense	Profit
January	900.00	1050.00	-150.00
February	1100.00	990.00	110.00

Figure 4-10: Right-justified tabbed text

If you wanted to achieve this type of alignment using a font that wasn't monospaced, you would need to calculate positions of the substrings of your text and draw the substrings yourself. Alternatively, you could use a `DataGridView` or some other more advanced control.

Displaying Text Using a Brush

So far, you've seen examples that use a black brush to draw text strings. However, you can use *any* type of custom or prebuilt `Brush` object, including any of the `Brush` objects described in [Chapter 3](#). Consider the following example:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Font f = new Font("Times New Roman", 48, FontStyle.Bold);
    HatchBrush hb = new HatchBrush(HatchStyle.Cross,
        Color.White, Color.Black);
    g.DrawString("Crazy Crosshatch", f, hb, 0, 0);
    f.Dispose();
}
```

This example creates a `HatchBrush`, and then draws text using that brush. (If you try it, don't forget that the `HatchBrush` class is in the `System.Drawing.Drawing2D` namespace.) When run, the output looks like [Figure 4-11](#).

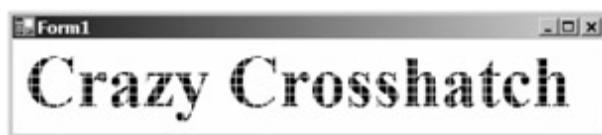


Figure 4-11: Text drawn with crosshatch brush

Previous

Next

Previous

Next

Finding Available Fonts

You may want to be able to find out what fonts are installed on the local machine. In fact, this is quite an important capability if your application is going to be installed on different computers, unless fonts are included as part of the installation.

You can check for available fonts by using the `InstalledFontCollection` class (which is part of the `System.Drawing.Text` namespace). The following code gets a list of all of the installed font families and writes the list to the console window (so, if you try it, you need to ensure the Output Type is set to Console Application in your application's Properties).

```
FontFamily[] fontFamilies;
InstalledFontCollection installedFontCollection =
    new InstalledFontCollection();

fontFamilies = installedFontCollection.Families;

for (int i = 0; i < fontFamilies.Length; ++i)
    Console.WriteLine("FontFamily name: " + fontFamilies[i].Name);
```

Here's a portion of the output that I get in the console window when I run this on my own machine:

```
FontFamily name: Arial
FontFamily name: Arial Black
FontFamily name: Arial Narrow
FontFamily name: AvantGarde Bk BT
FontFamily name: AvantGarde Md BT
FontFamily name: Book Antiqua
FontFamily name: Bookman Old Style
FontFamily name: Century Gothic
...
```

Another technique that you can use to get a list of font families is to call the `GetFamilies` method of the `FontFamily` class (passing an instance of the `Graphics` class as an argument to the method), like this:

```
Graphics g = e.Graphics;
FontFamily[] families = FontFamily.GetFamilies(g);

foreach (FontFamily family in families)
    Console.WriteLine("FontFamily name: " + family.Name);
```

This code will provide a list of font families that are available for the specified graphics context. The output of this code is identical to that of the previous example.

Now let's move on to look at metrics of fonts and the component pieces of fonts. This is a subject that will help you to gain more control over where you write text on your drawing surfaces.

 Previous

Next 

 Previous

Next 

Using Font Metrics for Precise Text Placement

In a discussion of how the .NET Framework handles font metrics, the first thing you need to know is the component pieces of a font. So, we'll begin with a description of each of the main font metrics.

Understanding Font Metric Components

The four main metrics that we are concerned with are the *ascent*, *descent*, *leading*, and *font height*. [Figure 4-12](#) shows each of these metrics.

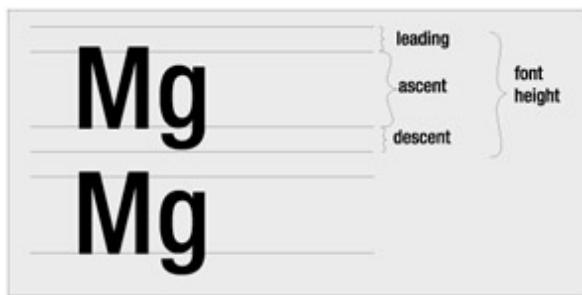


Figure 4-12: Metrics of a font

Note The term *leading* is pronounced "ledding." It is so-called because typographers used to use lead strips between lines of characters on a printing press. Leading refers to the line spacing—the empty space between consecutive lines.

Sometimes typographers use a different model, in which the ascent is broken down into the *x height* and the *ascender*, and the descent is referred to as the *descender*. [Figure 4-13](#) illustrates this model.



Figure 4-13: A different font metrics model

The first model (shown in [Figure 4-12](#)) is the one used by the .NET Framework, and the one that we will use in this book.

To really understand how to measure text, we need to look at one more metric: the *em* (or *em height*). Contrary to what you might think, the em height is not (necessarily) the height of the *M* character. Rather, an em is a unit of measurement defined as the point size of the font. Thus, if you're using a 12-point font, the length of the em unit is 12 points.

The em is also used to define the height and width of something called the *em square*. This is a term used in typography to describe the grid on which the typographer designs the glyphs of a font. The em square is a square grid that measures 1 em by 1 em in *design units*. Design units are the smallest units that typographers use to design their fonts. A design unit has no intrinsic relationship to any real-world measurement. In the case of TrueType fonts, there are typically 2048 design units in 1 em. The typographer lays out the glyphs on the grid using a variety of abstractions to describe the arcs and lines that make up the shape of the glyph.

Note Sometimes design units are called *font units*, *em units*, or *grid units*. In .NET, they are called design units.

In .NET, you need to use the em height to convert between design units and pixels when calculating the metrics of fonts. In GDI+ (although not in GDI), text can be resolution-independent. This means that the forms and windows that you build using GDI+ can look the same at all resolutions. Design units are key to accomplishing resolution independence.

Getting the Font Metrics of a Font Family

Sometimes when you're building custom controls, you are content to allow GDI+ handle the placement of text. In these cases, such as in the examples you've seen so far in this chapter, you simply supply a bounding rectangle for the text, and GDI+ positions the text inside the rectangle. However, in other cases, you may want to position the text explicitly and precisely. For example, you may want the baseline of the text to line up visually with some other part of your control, or you may want to center the text vertically and precisely relative to a horizontal line. In either of these cases, you will need to get detailed

To demonstrate getting font metrics, let's look at an example in which we ask GDI+ to tell us various different font metrics—ascent, descent, em height, and line spacing (leading)—for three different fonts. We can query the `FontFamily` class for these details and use a format string (like the one we used earlier in this chapter) to display them to the console window:

```
// Create the format string
String formatString = "{0,-16}{1,8}{2,9}{3,10}{4,14}";

// Write the first line of the table
Console.WriteLine(formatString, "Font Family Name", "Ascent", "Descent",
                  "EmHeight", "Line Spacing");

// Write font metrics for Courier New font family
FontFamily ff = new FontFamily("Courier New");
Console.WriteLine(formatString, ff.GetName(0),
                  ff.GetCellAscent(FontStyle.Regular),
                  ff.GetCellDescent(FontStyle.Regular),
                  ff.GetEmHeight(FontStyle.Regular),
                  ff.GetLineSpacing(FontStyle.Regular));

// Write font metrics for Arial font family
ff = new FontFamily("Arial");
Console.WriteLine(formatString, ff.GetName(0),
                  ff.GetCellAscent(FontStyle.Regular),
                  ff.GetCellDescent(FontStyle.Regular),
                  ff.GetEmHeight(FontStyle.Regular),
                  ff.GetLineSpacing(FontStyle.Regular));

// Write font metrics for Times New Roman font family
ff = new FontFamily("Times New Roman");
Console.WriteLine(formatString, ff.GetName(0),
                  ff.GetCellAscent(FontStyle.Regular),
                  ff.GetCellDescent(FontStyle.Regular),
                  ff.GetEmHeight(FontStyle.Regular),
                  ff.GetLineSpacing(FontStyle.Regular));
```

When you run this code, it will output the following:

Font Family Name	Ascent	Descent	EmHeight	Line Spacing
Courier New	1705	615	2048	2320
Arial	1854	434	2048	2355
Times New Roman	1825	443	2048	2355

The values that we've retrieved from the `FontFamily` here are expressed in terms of design units. As you can see, each of these three TrueType fonts has the following metrics:

- Designed using 2048 design units to 1 em
- An ascent of roughly 3/4 em (though each is a little different)
- A descent of roughly 1/4 em (though each is a little different)
- Line spacing of a little over 1 em (though each is a little different)

Converting Font Metrics into Physical Measurements

The values that we've just retrieved are dependent only on the font family, not on the font. So, for example, in terms of design units, the font metrics of 12-point Arial are the same as 26-point Arial. This isn't surprising, because different-sized instances of the same font family come from the same design, and therefore use the same glyphs that the typographer designed using the 2048-units-per-em grid.

If you want to use these values to achieve precise positioning of your text, you need to convert them from design units to the current graphics unit of your drawing surface, using the `em` size of the font family instance in which the text is to be drawn. This maps the relative sizes of the font metrics to actual measurements on your drawing surface.

In the next example, we're going to work out how to perform this conversion. Let's consider how we can do this. First, recall that when you construct a font, you pass the em size for the desired font as an argument to the `Font` constructor, like this:

```
Font f = new Font("Times New Roman", emSize);
```

By default, the desired em size is assumed to be in units of `GraphicsUnit.Point`. So, if you want a 12-point Arial font, you could use this:

```
Font f = new Font("Arial", 12);
```

This gives you an instance of the Arial font family whose em size is 12 points. As it happens, you're not obliged to use a `Point` measurement to specify the em size of the font. You can construct an instance of a font using any of the available `GraphicsUnit` members. In general, if you didn't want to use the `Point` measurement, you would use the same graphics unit as you have set for your drawing surface. In [Chapter 9](#), we'll discuss setting the graphics unit for a drawing surface.

With any instantiated font, you can use the `Font.Size` property to get the em size of the font in terms of the graphics unit for the font. Since we know the em size of the font *both* in terms of the graphics units *and* in terms of the design units, we can easily convert from design units to graphics units using this formula:

$$\text{Metric in GraphicsUnit} = \text{Metric in Design Units} \times \text{emSize in GraphicsUnit} / \text{emSize in Design Units}$$

Using Metrics for Alignment

You can see the measurement conversion in action in the following code. Remember the point of the exercise is to use these measurements to get precise alignment between the text string and other elements on the page. So, in the following example, we'll draw a couple of lines of text, and then place a few well-chosen lines at precise positions in relation to the text strings.

This is a longer example than previous ones, so we'll look at it in parts. The first task of interest is to create a font family, and then create a font instance (with a 24-point `emSize`):

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create the font family
    FontFamily ff = new FontFamily("Times New Roman");

    // Create the font, passing 24f as the emSize argument
    float emSizeInGU = 24f;
    Font f = new Font(ff, emSizeInGU);
```

Now that we have a font family and a font, we'll prepare for the design-unit-to-graphics-unit conversion. To do this, we need to get the values for the `emSize`, ascent, descent, and line spacing in terms of design units. Recall that these are the same for every font in the font family, so we get these values from the `FontFamily` object:

```
// Get the design unit metrics from the font family
int emSizeInDU = ff.GetEmHeight(FontStyle.Regular);
int ascentInDU = ff.GetCellAscent(FontStyle.Regular);
int descentInDU = ff.GetCellDescent(FontStyle.Regular);
int lineSpacingInDU = ff.GetLineSpacing(FontStyle.Regular);
```

Now we can use our conversion formula to convert the ascent, descent, and line spacing from design units to graphics units. This tells us the size of these three metrics, in points, for the 24-point instance of this font:

```
// Calculate the GraphicsUnit metrics from the font
float ascentInGU      = ascentInDU * (emSizeInGU / emSizeInDU);
float descentInGU     = descentInDU * (emSizeInGU / emSizeInDU);
float lineSpacingInGU = lineSpacingInDU * (emSizeInGU / emSizeInDU);
```

For the sake of interest, we'll write these values to the console window (using the format string technique

```
// Output the metrics to the console
Console.WriteLine("emSize = " + emSizeInDU + " DesignUnits");
Console.WriteLine("emSize = " + emSizeInGU + " GraphicsUnits");
string format = "{0,-16}{1,12}{2,16}";
Console.WriteLine(format,
    "Font Metric", "DesignUnits", "GraphicsUnits");

Console.WriteLine(format,
    "Ascent", ascentInDU, ascentInGU);
Console.WriteLine(format,
    "Descent", descentInDU, descentInGU);
Console.WriteLine(format,
    "Line Spacing", lineSpacingInDU, lineSpacingInGU);
```

Now, the proof of the pudding is in the eating, so we'll draw some text strings and some lines and see how well they align. First, we'll draw two simple lines of text using our 24-point font. In the following code, the first two lines create two points that will form the top-left corners of the rectangles containing the two lines of text (they are a distance of `f.Height` apart). The next two lines draw the text strings.

```
// Draw two lines of the text string
PointF textOrigin = new PointF(20, 20);
PointF nextLineOrigin = new PointF(textOrigin.X,
    textOrigin.Y + f.Height);

g.DrawString("AxgQ", f, Brushes.Black, textOrigin);
g.DrawString("AxgQ", f, Brushes.Black, nextLineOrigin);
```

Now let's try to use the results of our conversion to draw some lines that are precisely aligned to the text. Specifically, we'll draw horizontal lines to mark the following:

- The top of the leading (the line of the origin)
- The top of the ascent (the highest reach of the tallest characters)
- The bottom of the ascent (the baseline)
- The bottom of the descent (the lowest reach of the characters with tails)

First, here's the code for drawing the two lines at the top of the leading:

```
// Draw a line at the textOrigin
int lineLen = 100;
g.DrawLine(Pens.Blue,
    textOrigin,
    new PointF(textOrigin.X + lineLen, textOrigin.Y));
g.DrawLine(Pens.Red,
    nextLineOrigin,
    new PointF(nextLineOrigin.X + lineLen, nextLineOrigin.Y));
```

We've set the line length to 100 pixels. The top lines will be blue, and the bottom lines will be red. Each `DrawLine` method call draws a line between points, whose coordinates are calculated using the coordinates of the origin for the text string.

Now comes the real test: using `lineSpacingInGU`, `ascentInGU`, and `descentInGU` to draw similar lines for the other levels specified. First, here are the lines for the bottom of the ascent:

```
// Draw a line at the baseline
PointF p = new PointF(textOrigin.X,
    textOrigin.Y + lineSpacingInGU);

g.DrawLine(Pens.Blue, p,
    new PointF(p.X + lineLen, p.Y));

p = new PointF(nextLineOrigin.X,
    nextLineOrigin.Y + lineSpacingInGU);
g.DrawLine(Pens.Red, p,
    new PointF(p.X + lineLen, p.Y));
```

Next are the lines for the top of the ascent:

```
// Draw a line at the top of the ascent
p = new PointF(textOrigin.X,
                textOrigin.Y + lineSpacingInGU - ascentInGU);
g.DrawLine(Pens.Blue, p,
           new PointF(p.X + lineLen, p.Y));

p = new PointF(nextLineOrigin.X, nextLineOrigin.Y +
               lineSpacingInGU - ascentInGU);
g.DrawLine(Pens.Red, p, new PointF(p.X + lineLen, p.Y));
```

And finally, these are the lines for the bottom of the descent:

```
// Draw a line at the bottom of the descent
p = new PointF(textOrigin.X,
                textOrigin.Y + lineSpacingInGU + descentInGU);
g.DrawLine(Pens.Blue, p,
           new PointF(p.X + lineLen, p.Y));

p = new PointF(nextLineOrigin.X,
               nextLineOrigin.Y + lineSpacingInGU + descentInGU);
g.DrawLine(Pens.Red, p,
           new PointF(p.X + lineLen, p.Y));
}
```

When you run this, you see the two lines of text drawn to the window, as shown in [Figure 4-14](#). You can also see the two pairs of four horizontal lines that illustrate the various font metrics, precisely and correctly aligned to the text string itself. We were able to achieve this only when we took the design unit values of the metrics from the `FontFamily` object and converted them into the "real" measurements that relate to this particular (24-point) font instance.

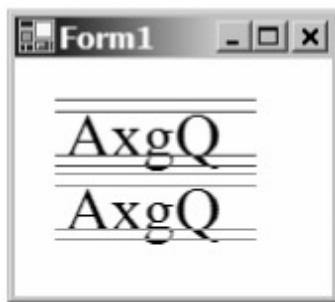


Figure 4-14: Detailed metrics of a font

Finally, let's check the output that was written to the console window:

```
emSize = 2048 DesignUnits
emSize = 24 GraphicsUnits
Font Metric      DesignUnits    GraphicsUnits
Ascent           1825          21.38672
Descent          443           5.191406
Line Spacing     2355          27.59766
```

It confirms that the `emSize` is 2048 design units, and that (in the case of this font) it's equal to 24 points. It also tells us the ascent, descent, and line spacing for 24-point Times New Roman, in points.

Previous

Next

Previous

Next

Controlling the Quality of Text

When you use GDI+ to draw text, you are given quite a bit of control over the visual quality of the text. To specify the desired level of quality, you can use the `TextRenderingHint` property of the `Graphics`

class. You can set this property to any value from the `TextRenderingHint` enumeration (which is part of the `System.Drawing.Text` namespace), as shown in [Table 4-3](#).

Table 4-3: Text Rendering Hints

Value	Description
<code>AntiAlias</code>	Anti-aliased characters are drawn without stem-width hinting.
<code>AntiAliasGridFit</code>	Anti-aliased characters are drawn with stem-width hinting.
<code>ClearTypeGridFit</code>	ClearType anti-aliased characters are drawn with stem-width hinting. This is the slowest to process, but with the best appearance. There is no option to draw using ClearType while not using stem-width hinting.
<code>SingleBitPerPixel</code>	Characters that are not anti-aliased are drawn <i>without</i> stem-width hinting. This is the fastest to process, but with the lowest-quality appearance.
<code>SingleBitPerPixelGridFit</code>	Characters that are not anti-aliased are drawn <i>with</i> stem-width hinting.
<code>SystemDefault</code>	The system's default smoothing setting is used. For most purposes, it is recommended that you use this setting.

Adjusting Stem Widths

When drawing very large and very small text in a given font, it is not enough to just make the glyphs bigger or smaller. The widths of certain parts of the characters will look out of proportion if you simply resize them to extreme sizes. To solve this problem, the font description often contains *hints* that tell the rendering engine about how to change the widths of certain parts of the characters, so that the font looks good at very large sizes and very small sizes.

Some of the `TextRenderingHint` values shown in [Table 4-3](#) tell GDI+ to adjust *stem widths* as appropriate when drawing in much larger fonts. (The *stems* of a glyph are the principal strokes, so *J* has one stem and *H* has two.) It takes more CPU time to draw with stem-width hinting, but on today's fast computers, this is not really noticeable.

Setting the Anti-Aliasing Method

As detailed in [Chapter 2](#), *anti-aliasing* is a technique that is used to get a better approximation of graphical operations like drawing curves, diagonal lines, and text. As shown in [Table 4-3](#), five different values, plus the system default, are for anti-aliasing text. The five different options can be roughly approximated as three methods:

- No anti-aliasing
- Anti-aliasing
- Anti-aliasing using ClearType font features

Characters drawn *without* any anti-aliasing look like those shown in [Figure 4-15](#).

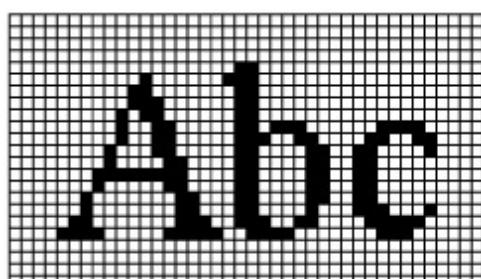


Figure 4-15: Text without anti-aliasing

The standard way of anti-aliasing involves using different lighter shades to draw the pixels that are on the boundaries of the glyphs that make up the characters. This works best on CRT displays. [Figure 4-16](#) shows the same characters that are in [Figure 4-15](#), but with *standard* anti-aliasing.

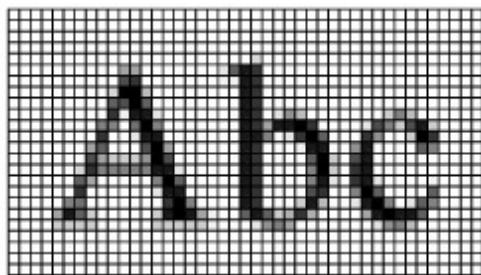


Figure 4-16: Text with standard anti-aliasing

ClearType anti-aliasing is a technology that is patented by Microsoft and gives significantly clearer text than standard anti-aliasing. It works only on color LCD screens, and only in Windows XP. It uses the fact that each pixel on a color LCD screen is actually made up of three narrow vertical pixels: one each of the colors red, green, and blue. By setting colors at the edge of characters appropriately, ClearType anti-aliasing sets the brightness of these tall, narrow pixels, creating a smoother appearance to the text.

Although it is a little difficult to show the colored pixels in a monochrome illustration, [Figure 4-17](#) contains an image from a digital microscope that shows the vertical colored pixels that make up an LCD screen.

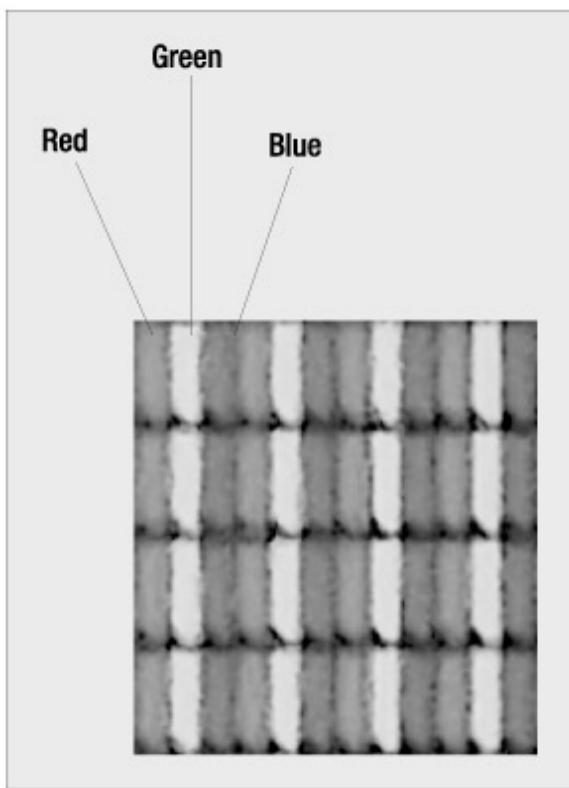


Figure 4-17: Thin, tall LCD pixels

Unaided, the human eye cannot perceive these small individual pixels. If all three pixels are fully on (that is, as bright as they can be), then the combined effect appears to be white.

The following three figures show how the same character is treated differently by the three basic levels of anti-aliasing. [Figure 4-18](#) shows a lowercase letter c without any anti-aliasing. As you can see, the narrow vertical pixels are turned on and off three at a time.

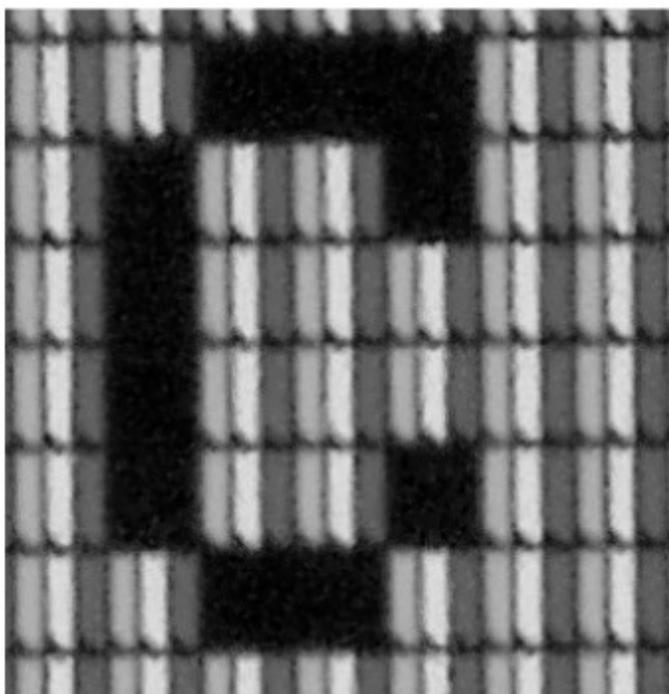


Figure 4-18: A c without anti-aliasing

Note If you want to compare the color versions of these images, you can download them, along with the source files for this book, from the Downloads section of the Apress web site (www.apress.com).

[Figure 4-19](#) shows the same character using standard anti-aliasing. In this case, you can see that the standard anti-aliasing modifies the brightness of the narrow vertical pixels three at a time. When the standard anti-aliasing modifies three of these tall, narrow pixels, it is modifying the brightness of a single logical pixel on the screen.

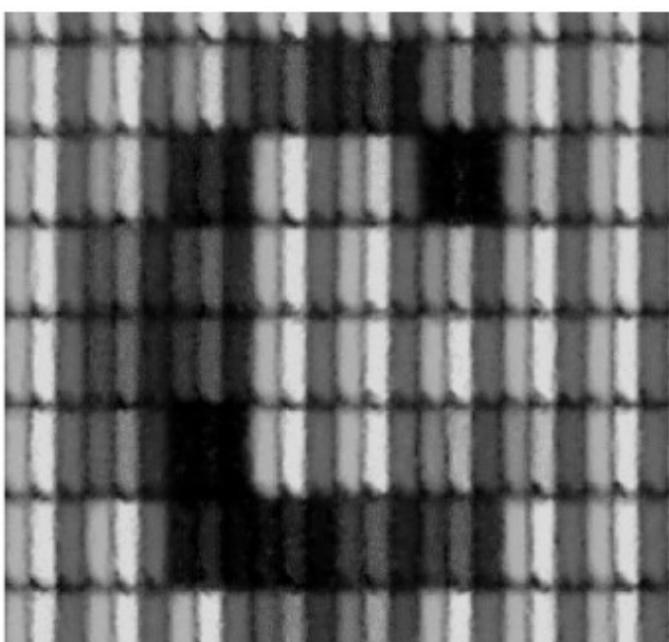


Figure 4-19: A c with standard anti-aliasing

[Figure 4-20](#) shows anti-aliasing using ClearType. The anti-aliasing algorithm sets the brightness of the individual red, green, and blue pixels. It doesn't completely turn the narrow vertical pixels on or off; instead, it sets the brightness of them to give the best appearance of the text.

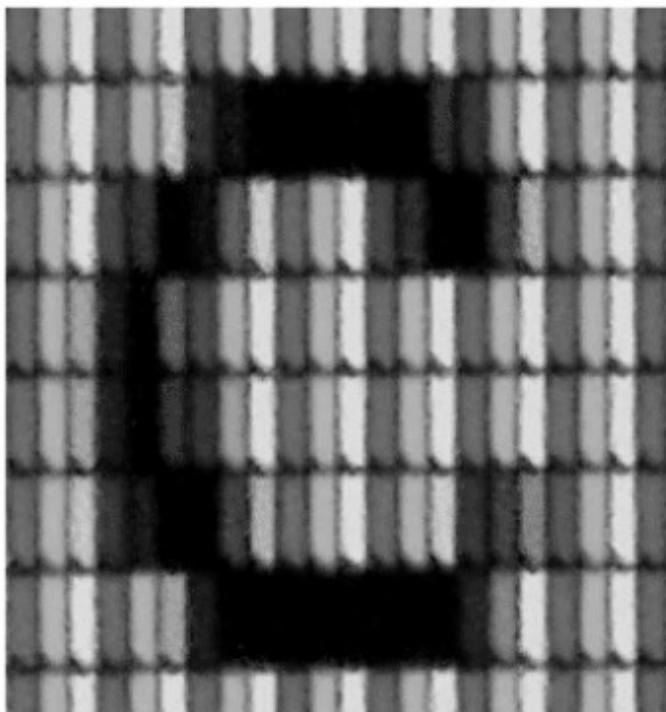


Figure 4-20: A c with ClearType anti-aliasing

In Windows XP, users can specify the required level of anti-aliasing (none, standard, or ClearType) via the Display Properties dialog box. In the code for the control, if the `TextRenderingHint` property is set to `SystemDefault`, then GDI+ draws according to preferences set in this dialog box. This means that if you draw your text without explicitly setting the anti-aliasing option in your code, text is drawn according to the advanced effects settings in the display properties. However, you can override the user's preferences by explicitly setting `TextRenderingHint` to one of the other values.

In Windows 2000, the Effects tab of the Display Properties dialog box includes a check box that allows the user to specify that edges of screen fonts should be smoothed. However, this setting doesn't have any effect on text that you draw using GDI+. If you want to use standard anti-aliasing, you must set the `TextRenderingHint` property to `AntiAlias`, as shown in the following example. (ClearType anti-aliasing is not available with Windows 2000.)

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.TextRenderingHint = TextRenderingHint.AntiAlias;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawString("Hello", this.Font, Brushes.Black, 0, 0);
}
```

CLEARTYPE HARDWARE AND PERFORMANCE ISSUES

ClearType is currently implemented only for Windows XP and for screens whose left-to-right ordering of the subpixels is red—green—blue. This is a hardware issue. Most LCD screens are ordered red—green—blue, but some are not. The best way to determine if your LCD screen is ordered red—green—blue is to turn on ClearType anti-aliasing and see if the resulting image is better.

In addition, at the time of writing, ClearType anti-aliasing is not supported for text strings that are drawn in any direction *other* than horizontally. The reason for the current lack of support is simple: vertically aligned ClearType anti-aliased text would require a completely different algorithm from the one included in the current implementation. It is conceivable that Microsoft could implement ClearType anti-aliasing for vertically aligned text (or indeed, for text aligned at any angle) in the future.

My personal opinion is that ClearType anti-aliasing is very good. It gives a better appearance than text rendered using either no anti-aliasing or standard anti-aliasing. In particular, when using

ClearType antialiasing on a laptop that has a native resolution of approximately 120 DPI, aliasing artifacts are almost completely eliminated.

However, there is a noticeable speed trade-off. On a Pentium II laptop that is running at 300 MHz, the drawing speed is visibly slower when using the ClearType anti-aliasing, although it is adequate. On a somewhat faster computer, such as a Pentium III 1 GHz laptop, GDI+ draws using ClearType so quickly that there is no noticeable drawing time.

The best solution may be to draw your text without explicitly setting the anti-aliasing option in your code, so that the text is drawn according to the advanced effects settings in the Windows XP Display Properties dialog box. Users can then decide whether or not to enable ClearType anti-aliasing, and your text drawing code will conform to their preference.

 Previous

Next 

 Previous

Next 

Drawing Device-Independent Graphics

One of the main features of GDI+ is that it enables you to draw device-independent graphics. The most difficult problem to overcome when drawing device-independent graphics is the issue of dealing with a wide range of display resolutions. One of the tools at your disposal is the `Graphics.PageUnit` property. The `Graphics.PageUnit` is the unit of measure of the drawing surface. You can set this unit of measurement to `Inch`, `Pixel`, `Display (1/96 inch)`, `Millimeter`, and so on.

One approach that makes it easy to coordinate the drawing of graphics and the drawing of text is to set the unit of measurement of the drawing surface (the `Graphics.PageUnit`) and the unit of measurement of the font (an argument to the constructor) to the same value. Both settings are of type `GraphicsUnit`, so you can use the same `GraphicsUnit` enumeration to achieve this.

So, for example, suppose you are drawing a graphic such as picture of a ruler, which is marked with inch marks along its edge. You can set the `Graphics.PageUnit` property to `GraphicsUnit.Inch`, and then express the coordinates of the drawing operations in terms of inches. You can set the unit of measure of the font to `GraphicsUnit.Inch`, and then express the size of the text in terms of inches.

Consider the following code:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.GraphicsUnit = GraphicsUnit.Inch;

    Pen p = new Pen(Color.Black, 1/96f);
    Font f = new Font("Times New Roman", 16);
    String s = "Abc";
    SizeF sf = g.MeasureString(s, f);

    g.DrawRectangle(p, 1, 1, sf.Width, sf.Height);
    g.DrawString(s, f, Brushes.Black, 1, 1);

    f.Dispose();
    p.Dispose();
}
```

First, we set the unit of measurement of the drawing surface to `Inch`. Then we express the pen width in terms of inches. Given that GDI+ defines the resolution of windows to be 96 DPI, it makes sense to create a pen with a width of 1/96 inch.

Note The resolution of windows does not depend on the setup of the user's computer. GDI+ defines all screens to be 96 DPI, regardless of the actual resolution of the device itself.

Then we create a font. After that, when we use the `DrawString` method to draw the string, we specify

the drawing location in terms of inches, too, marking the position (1, 1) (with the coordinates measured in inches). But note that when we created our font, we expressed the size in terms of *points*, not inches.

When you run this code, you will see the string drawn in 16-point Times New Roman, 1 inch from the top of the window and 1 inch from the left of the window, as shown in [Figure 4-21](#).



Figure 4-21: Text drawn 1 inch over and 1 inch down

Clearly, the choice of unit of measure for the coordinate system for the drawing surface is independent of how you express values like the font size. But you can make them the same by setting the `PageUnit` property of the `Graphics` object, and then passing in the same `GraphicsUnit` as an argument to the `Font` constructor, like this:

```
Pen p = new Pen(Color.Black, 1/96f);
Font f = new Font("Times New Roman", 0.5f, GraphicsUnit.Inch);
String s = "Abc";
SizeF sf = g.MeasureString(s, f);
```

Now the font size is expressed in terms of inches. This example draws text that is 1/2 inch high, 1 inch from the top of the window, and 1 inch from the left of the window, as shown in [Figure 4-22](#).

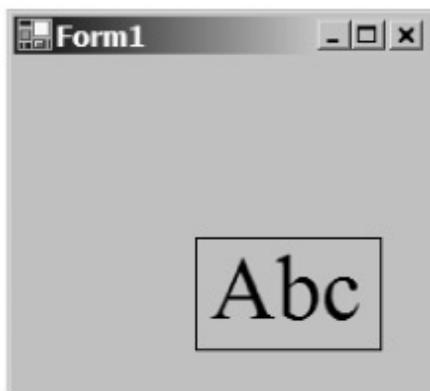


Figure 4-22: Text that is 1/2 inch high

Previous

Next

Previous

Next

Summary

We've covered a lot of ground in this chapter, establishing some important definitions and illustrating useful techniques. The chapter began by introducing the Unicode character encoding system—a system that maps every typewritten character to a unique 16-byte code. Then we established the difference between five crucial concepts: character, glyph, font, font family, and typeface. A *glyph* is a graphic

symbol that represents a character, a *font* is a set of glyphs, and a *typeface* is a set of fonts that are identical except in size. A *font family* is a collection of similar typefaces, which differ only in (bold or italic) style.

We took a look at the process of specifying font families and creating a font from a given font family. The `Font` class has 13 different constructors, and over the course of the chapter, you saw quite a few of them in use. They help you to achieve simple effects like bold, italic, underline, and strikeout, as well as a range of more complex effects.

You saw an example that uses the system's default font and examined more closely how you can use the `DrawString` method to draw text to your drawing surface. Typically, you need to specify things like the text string, a font, and a brush. You must also use the coordinate system of the drawing surface, of course, to specify the desired position of the text on the drawing surface.

We looked at the rectangle that bounds a text string. It's important to be able to control the dimensions of the rectangle in which a text string is to be placed. You often need to be able to dictate the maximum width that the text string occupies or the maximum number of lines it uses. You saw how you can use the `Graphics.MeasureString` method to get a `Size` object that tells you how a string is likely to be rendered, and to use the dimensions of the `Size` object actually to draw the text string.

You also saw how to force line breaks using the `\n` character. We worked through examples that used the `StringFormat` class to create a whole range of different formatting effects: orienting the text vertically, drawing with and without line wrap, and aligning text horizontally and vertically. We also used the `StringFormat` class, with the `String.Format` property, to apply standard formatting to a number of text strings. Our example used this technique to build a table effect.

Then we turned our attention to fonts. You saw that you can use the `InstalledFontCollection` class to establish which fonts are loaded on a system. Next, we began a careful study of font metrics. You saw that a font size is determined by something called the *em* (for example, for a 12-point font, the length of the em unit is 12 points). We looked at design units and resolution, and how the precise placement of text on a page (relative to other elements) can be achieved by a conversion between the font metrics of the font family (in design units) and the units used to draw the font instance.

Next, we took a look at how anti-aliasing technology has reached new levels of sophistication in GDI+. Using ClearType anti-aliasing, you can achieve a very good quality of text using LCDs). Finally, you saw the advantages of using the same unit of measurement for the drawing surface and for the font.

We haven't covered every last detail of drawing text. GDI+ has capabilities for advanced text drawing that are mostly of interest to developers writing advanced typography applications, such as page layout programs, word processors, and the like. However, we have covered sufficient material to fully satisfy your needs for most custom controls.

 Previous

Next 

 PreviousNext 

Chapter 5: Images

Overview

Whether you are developing a Windows application or an ASP.NET application, the proper use of some well-designed images will considerably enhance its appeal. For example, if you're designing a Windows application, you could use GDI+ to display an image in a button, within a cell of a spreadsheet, or as part of a node in a hierarchical tree control. If you're building an ASP.NET web application, you could customize a web page for a specific visitor by populating it with images that are generated dynamically on the web server using GDI+. It's easy to imagine how you could employ this technique in a number of different types of applications. For example, you could use it in an application for charting stock prices or in one that shows custom maps to the user, with the maps being generated dynamically according to the user's needs.

In this chapter, we will focus on two GDI+ classes that allow you to handle images: the `Image` class and the `Bitmap` class. We'll start with a brief overview of image handling in GDI+ and the various types of bitmap files that you can use. We will then investigate the functionality that these classes provide and how you can achieve the following:

- **Display images:** Some applications just need to be able to *display* images. For example, a real-estate application might display an image of the property for sale (and make the image available for printing). We'll investigate the GDI+ options for controlling the size, scale, resolution, and quality of images.
- **Manipulate images:** Some applications need to be able to *manipulate* images. You'll see how to crop, skew, rotate, and clone images, and how to create thumbnail images.
- **Load and save images:** Some applications need to be able to store their drawing work to some kind of file or other permanent storage, or to load files from disk for use at runtime. You'll see how to perform both of these operations.

Moreover, some applications *don't* draw images directly to the screen. Instead, they draw the object "abstractly"—in memory—and draw the prepared image to the screen (or to disk) when it's complete. You may prepare the image dynamically on the web server, at runtime, and then retrieve it to be displayed on a web page. Preparing images off-screen can give a considerable performance benefit, especially for complex, layered graphics. You'll see how GDI+ allows you to draw images abstractly by using a `Bitmap` object as the drawing surface.

We'll then move on to take a brief look at the use of animated graphics. Certain types of animations, such as GIF animations, are stored in image files. We'll cover how to start and control a GIF animation in a Windows application.

It's also useful to be able to load and save images in various formats, and know how to save an image to a different format from the one in which it was supplied. In particular, when building images to be included as part of a web page, you want to be very specific about exactly what type of image you build. Different image types have difference storage requirements, and therefore require more or less time to transmit to a browser. We'll finish off the chapter with a look at how you can use GDI+ to load images in one format and save them in another format.

 PreviousNext  PreviousNext 

An Overview of Image Handling

We can break down the broad task of image handling into a number of fairly general but well-defined subtasks:

- *Loading* the image from a file or other source, or creating a new image

- *Displaying* or rendering the image to screen, or to some other drawing surface
- *Manipulating* the image (making changes to the image in memory)
- *Saving* the image-in-memory to a file or other permanent storage

GDI+ provides a handsome functionality set that allows you to perform all of these tasks for many different types of images. This section introduces the GDI+ classes that provide this functionality. We'll also look at the different types of bitmaps that you can work with and the pixel formats that are supported by a bitmap in GDI+.

The Image, Bitmap, and Metafile Classes

GDI+ provides two classes that you can use to represent images: the `Bitmap` class and the `Metafile` class. Both of these classes inherit from the `Image` class, and therefore share a certain amount of functionality, but in many ways, they are also very different.

The Image Class

The `Image` class is an abstract class, which means that you cannot create `Image` objects directly. You can create instances of only the `Bitmap` and `Metafile` classes that inherit from it.

You can use an `Image` object (or, to be more accurate, a `Bitmap` or `Metafile` object) for two purposes: to draw an image to a drawing surface and to use an image as a drawing surface. The abstract `Image` class encapsulates some basic functionality that is inherited by both the `Bitmap` and `Metafile` classes. This functionality includes inherited properties such as the `Height` and `Width` of the image and the `Palette`, and a number of methods that either tell you about the image (for example, whether it contains alpha information, and whether the image uses a color map) or perform operations (for example, to `Clone` or `Save` the image).

The Bitmap Class

The `Bitmap` class provides functionality for handling bitmaps. A *bitmap* is a raster-based image. As you'll recall from [Chapter 2](#), this means that it is an image composed of square pixels arranged in a two-dimensional grid. Each pixel in the grid has its own color. The overall effect of all these colored pixels in the grid is of an image.

So, the `Bitmap` class encapsulates the basic functionality that allows you to control the fundamental characteristics of raster-based bitmaps, such as size, resolution, color depth, whether the bitmap contains alpha information, and whether it uses a color map.

The image representation itself is in the form of an array of bits, which dictate the color of each pixel. Each pixel's color is represented by a fixed number of bits, as you'll see later on, in the "[Pixel Format](#)" section.

A `Bitmap` object is actually a representation of a bitmap image maintained *in memory* at runtime. You can use the `Bitmap` object's functionality to load an image from disk or create a completely new image. Once you've done that, you can use the functionality to perform many other tasks on the image in memory, such as the following:

- Manipulate the image (for example, by drawing more elements to it or by cropping it).
- Set or get individual bits of the image or other properties of the image such as its size.
- Draw the image to another drawing surface.
- Save it, in a specified format (which may or may not require conversion), to a file or a stream.

Let's take a look at a very simple example. We'll create a `Bitmap` object and use it to load a bitmap from a file into memory. Then we'll use the `Graphics` class to display the image in memory onto our drawing surface.

Note Before you can run this example, you'll need to put your image file in the directory where the application will run. If you're running the example in debug mode, then place `rama.jpg` into the `/bin/debug` subdirectory of your project directory. (Alternatively, you can provide the full path to the image file.)

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.DrawImage(bmp, 0, 0);
}
```

Note The new `Bitmap` object we've created here is based on the specified JPEG file, `rama.jpg`.

You could use another type of file, if the appropriate encoder/decoder is available on the system. We'll discuss encoders and decoders in the "[Encoders and Decoders](#)" section later in this chapter.

When run, this code will draw the image to your form, with the upper-left corner of the image located at pixel (0, 0) in the client window, as specified in the `DrawImage` method call. [Figure 5-1](#) shows the result. (The image is of a very sweet, obedient dog, who was rescued from the gutter in India. His name is Rama.)



Figure 5-1: An image in a form

This is about the simplest possible application that you can use to read and display an image. As you would expect, you can have much more control over the display of the image than in this example, as you'll see in the "[Displaying Images](#)" section later in this chapter.

The Metafile Class

The `Metafile` class provides functionality for handling *vector images*. This class acts more like a recorder. As you use a `Metafile` object to perform drawing operations, the object records these operations in a queue. When you draw the `Metafile` object to a drawing surface, it replays the drawing operations, applying them one at a time to the drawing surface. If any active transformations (such as rotations, scaling, or translation) are applied to the drawing surface, the transformations are applied separately to each individual drawing operation.

In the future, metafiles will be important in the implementation of scalable icons for displays that have very high resolutions. For example, currently, certain graphical items in Windows operating systems are drawn in a hard-coded way with a fixed number of pixels, and yet are too small to be viewed comfortably

on LCD screens with very high resolution. A solution to this problem would be to replace the existing raster-based icons with metafile icons, described using graphical operations expressed in vectors, and scalable up or down in size to any extent.

However, in this book, we are concerned chiefly with raster images, and so we will focus almost exclusively on the `Image` class and the `Bitmap` class.

Types of Bitmaps

GDI+ supports the bitmap file formats listed in [Table 5-1](#). The table also lists the possible bits-perpixel storage formats and the type of compression supported by each. These characteristics are described in the following sections.

Table 5-1: Bitmap Types Supported by GDI+

Format	Description	Bits Per Pixel	Compression
BMP	Windows bitmap is a standard format used by Windows.	1, 4, 8, 15, 24, 32, 64	None
EXIF	Exchangeable Image File is a variation of the JPEG file format used by many digital cameras. It contains additional information about the image (such as the date taken and shutter speed) and about the camera (such as the manufacturer and model).	24	Lossy
GIF	Graphics Interchange Format is a common format for web pages. It is appropriate for screenshots and line drawings. In addition, GIF can contain multiple images that can be played as an animation.	1, 2, 4, 8	Lossless
JPEG	Joint Photographic Experts Group is another common format for web pages. It is best for pictures.	24	Lossy
PNG	Portable Network Graphics format is similar to GIF, but has more features. It supports greater color depth, and it enables progressive image display, which you might use when transmitting images over a slow network connection.	8, 24, 48 for color 1, 2, 4, 8 for grayscale	Lossless
TIFF	Tag Image File Format is an older format. It supports a wide variety of color depths and compression. TIFF files are often used in desktop publishing, faxing, 3D applications, and medical imaging.	Various	

Color Depth

When you store a bitmap in a persistent storage medium such as a file on disk, you often have the option to specify the color depth of the image; that is, the number of bits per pixel. As mentioned in [Chapter 2](#), if color is specified in RGB format, and you require 8 bits to specify the level of intensity of each of the red, green, and blue elements of the color, you need 24 bits of disk space to store the color information for a single pixel. If the image contains alpha information, another 8 bits per pixel are required.

Sometimes to save space, an image's color information is in the form of a *color map* (this is sometimes called a *color table* or a *color palette*). In this case, instead of storing the red, green, and blue values for each pixel, a bitmap stores an integer for each pixel. The integer is an index into the color map (and hence represents a color). If the total number of colors in the image is less than 256, the color table needs at most 256 colors in it, and hence the index for each pixel can be contained in a single byte (8 bits). Thus, you can get a significant reduction in file size compared to a bitmap that stores 24 or 32 bits per pixel. If the bitmap is a monochrome bitmap, using only two colors, you can use 1 bit per pixel for the color.

The decision as to whether to implement a color map for a bitmap (and hence restrict the number of colors and the size of the file) is influenced less these days by the amount of memory required to store the image, and more by the need to keep the images small enough that they can be transmitted efficiently over the Internet. JPEG files do not use color maps, but GIF files can. Color maps are best

suites for simple images that don't use a large number of colors, such as graphs and other line drawings.

Compression Formats

GDI+ supports many of the standard file formats for storing bitmaps. Many of these file formats will support lossy or lossless compression.

With *lossy* compression, some information may be lost in the compression, but for certain types of image, this lost information is indiscernible. Photographs lend themselves to lossy compression. There are many levels of lossy compression. The greater the compression, the more information is lost, and the further the compressed image deviates from the original.

If you use lossy compression on an image that has regular rectangular patterns, you will see an effect called *pixelation* around the lines. The lines will not look so neat, and you may see something similar to shadows of the lines, which is the pixelation effect.

Lossless compression takes advantage of the regularity of rectangular patterns within the image. Long horizontal or vertical runs of a single color can be represented in a parameterized manner, resulting in a significant reduction in file size. Traditionally, screenshots and computer-generated graphics are compressed effectively using lossless compression. However, with the advent of much fancier user interfaces (like those in Windows XP, with graduated title bars and controls drawn with gradients), lossless compression is becoming less and less effective.

Encoders and Decoders

When you're using a `Bitmap` object to manipulate an image, you don't really care about the format that GDI+ uses internally. You just interact with the image through the `Bitmap` object and allow GDI+ to worry about the storage details. However, when it comes to saving and loading images, you care very much about the format of the images. For example, if you're saving an image for consumption by another program, or with a certain storage requirement profile, you need to take control of the way in which the image is saved.

In GDI+, you use *encoders* to save images to files, and you use *decoders* to load images from files. Sometimes, when talking about encoders and decoders, we use the term *codecs*, which is a generic term for *both* encoders and decoders.

The `Encoder` and `Decoder` classes enable you to extend GDI+ to support any image format. The .NET Framework provides a selection of encoders and decoders for different image formats. And if you need an encoder or decoder that is *not* supplied by the .NET Framework, you have the facilities to write your own.

So, how do you find out what codecs are available on your system? You can use the `ImageCodecInfo` class, as shown in the following code.

Note `ImageCodecInfo` is in the `System.Drawing.Imaging` namespace, so to run this example, you need to add a `using` statement at the top of the file. If it's a console application, you'll first need to add references to `System.Drawing.dll` and `System.Drawing.Design.dll`.

```
// Get an array of available codecs
ImageCodecInfo[] availableCodecs;
availableCodecs = ImageCodecInfo.GetImageEncoders();
int numCodecs = availableCodecs.Length;

for (int i = 0; i < numCodecs; i++)
{
    Console.WriteLine("Codec Name = " + availableCodecs[i].CodecName);
    Console.WriteLine("Class ID = " + availableCodecs[i].Clsid.ToString());
    Console.WriteLine("Filename Extension = " +
        availableCodecs[i].FilenameExtension);
    Console.WriteLine("Flags = " +
        availableCodecs[i].Flags.ToString());
    Console.WriteLine("Format Description = " +
        availableCodecs[i].FormatDescription);
    Console.WriteLine("Format ID = " +
        availableCodecs[i].FormatID.ToString());
    Console.WriteLine("MimeType = " + availableCodecs[i].MimeType);
```

```
Console.WriteLine("Version = " +
    availableCodecs[i].Version.ToString());
Console.WriteLine();
}
```

The following is the output that I get when I run this code on my machine. When you run the code, you should get something similar (don't forget to change the Output Type to Console Application).

```
Codec Name = Built-in BMP Codec
Class ID = 557cf400-1a04-11d3-9a73-0000f81ef32e
Filename Extension = *.BMP;*.DIB;*.RLE
Flags = Encoder, Decoder, SupportBitmap, Builtin
Format Description = BMP
Format ID = b96b3cab-0728-11d3-9d7b-0000f81ef32e
MimeType = image/bmp
Version = 1
```

```
Codec Name = Built-in JPEG Codec
Class ID = 557cf401-1a04-11d3-9a73-0000f81ef32e
Filename Extension = *.JPG;*.JPEG;*.JPE;*.JFIF
Flags = Encoder, Decoder, SupportBitmap, Builtin
Format Description = JPEG
Format ID = b96b3cae-0728-11d3-9d7b-0000f81ef32e
MimeType = image/jpeg
Version = 1
```

```
Codec Name = Built-in GIF Codec
Class ID = 557cf402-1a04-11d3-9a73-0000f81ef32e
Filename Extension = *.GIF
Flags = Encoder, Decoder, SupportBitmap, Builtin

Format Description = GIF
Format ID = b96b3cb0-0728-11d3-9d7b-0000f81ef32e
MimeType = image/gif
Version = 1
```

```
Codec Name = Built-in TIFF Codec
Class ID = 557cf405-1a04-11d3-9a73-0000f81ef32e
Filename Extension = *.TIF;*.TIFF
Flags = Encoder, Decoder, SupportBitmap, Builtin
Format Description = TIFF
Format ID = b96b3cb1-0728-11d3-9d7b-0000f81ef32e
MimeType = image/tiff
Version = 1
```

```
Codec Name = Built-in PNG Codec
Class ID = 557cf406-1a04-11d3-9a73-0000f81ef32e
Filename Extension = *.PNG
Flags = Encoder, Decoder, SupportBitmap, Builtin
Format Description = PNG
Format ID = b96b3caf-0728-11d3-9d7b-0000f81ef32e
MimeType = image/png
Version = 1
```

From this output, you can see that on my machine, I have codecs for BMP, JPEG, GIF, TIFF, and PNG file types.

Pixel Format

Every image has an associated *pixel format*. The pixel format of the image specifies the color depth of the image, whether the colors are indexed with a color map, and whether the image contains alpha information.

When you read an image from a file, the file contains the pixel format of the image. When the image is held in memory, the pixel format is also held in memory. It's an integral part of the makeup of the image.

To find out about the pixel format of an image, you can use the `PixelFormat` property (this is a property of the `Image` class, and therefore inherited into the `Bitmap` and `Metafile` classes). When you create an image, you can specify your desired pixel format, or you can use a constructor that returns an image that matches the pixel format of a specified drawing surface. When you clone an image (that is, when you use an existing image to create a new one), you can specify the pixel format for the new cloned image explicitly, which means that you can change it if you want.

The `PixelFormat` property contains a value that belongs to the `PixelFormat` enumeration (which is part of the `System.Drawing.Imaging` namespace). The `PixelFormat` enumeration contains three different varieties of members, and we'll look at each of them here.

Available Pixel Formats

The set of 14 members shown in [Table 5-2](#) defines the supported values for the pixel format. The member names have been designed to summarize the pixel format they represent. First, all of the names are prefixed with the word `Format`. The next part of the name indicates the number of bits per pixel (that is, the amount memory that the format uses to store the color information for a single pixel)—like `8bpp` or `32bpp`. The next part describes the type of information that the bits represent:

- `Rgb` means that the bits-per-pixel storage is broken down into red, green, and blue components.
- `Argb` means that each pixel also has an alpha component.
- `PArgb` means that the red, green, and blue components are pre-multiplied according to the alpha component.
- `Indexed` means a color table is used.
- `Grayscale` means there is no color, but rather the specified number of shades of gray.

Table 5-2: Pixel Formats

PixelFormat Member	Color Depth Description
<code>Format24bppRgb</code>	8 bits of red, 8 bits of green, 8 bits of blue
<code>Format32bppRgb</code>	8 bits of red, 8 bits of green, 8 bits of blue, 8 bits not used
<code>Format48bppRgb</code>	16 bits of red, 16 bits of green, 16 bits of blue
<code>Format32bppArgb</code>	8 bits of red, 8 bits of green, 8 bits of blue, 8 bits of alpha
<code>Format64bppArgb</code>	16 bits of red, 16 bits of green, 16 bits of blue, 16 bits of alpha
<code>Format32bppPArgb</code>	8 bits of red, 8 bits of green, 8 bits of blue, 8 bits of alpha (red, green, and blue components are pre-multiplied according to the alpha component)
<code>Format64bppPArgb</code>	16 bits of red, 16 bits of green, 16 bits of blue, 16 bits of alpha (red, green, and blue components are pre-multiplied according to the alpha component)
<code>Format16bppRgb555</code>	32,768 shades of color (5 bits of red, 5 bits of green, 5 bits of blue, and 1 bit not used)
<code>Format16bppRgb565</code>	65,536 shades of color (5 bits of red, 6 bits of green, 5 bits of blue)
<code>Format16bppArgb1555</code>	32,768 shades of color (5 bits of red, 5 bits of green, 5 bits of blue, and 1 bit of alpha)
<code>Format1bppIndexed</code>	Two shades of color using a color table
<code>Format4bppIndexed</code>	16 colors using a color table
<code>Format8bppIndexed</code>	256 colors using a color table
<code>Format16bppGrayScale</code>	65,536 shades of gray

Note Pre-multiplication according to the alpha component is a technique that involves scaling the three color components by the alpha component *before* storing their values. If you intend to use

alpha blending (that is, to display two overlapping images, using the alpha components of the two images to create an effect in which they're both semitransparent and hence "blended" where they overlap), pre-multiplication saves a significant amount of processing time. Otherwise, pre-multiplication is not required. Note that if an image is stored with pre-multiplication and a low alpha value, colors may not show true if the image is subsequently displayed at full opacity.

Finally, if there is a sequence of integers (555, 565, or 1555) at the end of the member name, it represents the breakdown of red—green—blue (or alpha—red—green—blue) components in a situation where they're not split equally across the available bits per pixel.

Capabilities of an Image

If you want to test an image to determine whether it has particular capabilities, you can test for the `PixelFormat` enumeration members in the `PixelFormat` property:

- `Alpha`: Used to determine whether an image contains alpha information.
- `PAlpha`: Used to determine whether an image contains alpha information, and if so, whether the color components are pre-multiplied according to the alpha component.
- `Indexed`: Used to determine whether an image has a color table. Some operations, such as drawing with gradient brushes, are not allowed on images that have a color table.

To implement these tests, you "bitwise AND" the pixel format from the image with one of the above members. If the result is nonzero, you interpret that as an affirmative. In the following code, we create two new images: one with an alpha component and the other without. We then test the two images.

```
// Create two new bitmap images
Bitmap bmp1 = new Bitmap(100, 100, PixelFormat.Format32bppArgb);
Bitmap bmp2 = new Bitmap(100, 100, PixelFormat.Format24bppRgb);

// Test for alpha
bool b1 = ((bmp1.PixelFormat & PixelFormat.Alpha) != 0);
bool b2 = ((bmp2.PixelFormat & PixelFormat.Alpha) != 0);

// Output results to console window
Console.WriteLine("bmp1 has alpha?: " + b1);
Console.WriteLine("bmp2 has alpha?: " + b2);

// Clean up
bmp1.Dispose();
bmp2.Dispose();
```

The output of this code is as follows:

```
bmp1 has alpha?: True
bmp2 has alpha?: False
```

The Easy Option

Finally, there is one member of the `PixelFormat` enumeration that allows you to specify that you don't care what the pixel format is when you create an image:

- `DontCare`: Indicates that you don't care what pixel format the image has.

Later on, in the "[Cloning an Image](#)" section of this chapter, we will use this member when we clone an image. Effectively, it means that you allow GDI+ to choose the pixel format of the newly cloned image. If you don't have specific storage or image-translation requirements, and if you want a lossless uncompressed image, then you can use this member.

Displaying Images

In this chapter, you've already seen one simple example that loaded an image from disk into memory (using a `Bitmap` object), and then drew that image onto the drawing surface. The entire example was only a few lines long, so you could see this simple concept in action.

You can have much greater control over how an image is displayed by adding extra code to affect the size, scale, resolution, and interpolation of the image that appears on the drawing surface. In this section, we'll take a look at these issues and how they affect the image display.

Checking the Size and Resolution

In the first example in this chapter, we just loaded the image and displayed it, and left all the complicated stuff to GDI+. Here's the code we used:

```
Bitmap bmp = new Bitmap("rama.jpg");
g.DrawImage(bmp, 0, 0);
```

In order to process the image in memory and draw it to the drawing surface, GDI+ will have taken into account a number of factors: the image's (horizontal and vertical) size, the image's resolution, and the resolution of the drawing surface. All that took place behind the scenes.

Let's take a more active interest in what GDI+ is doing here. The bitmap image's size (its width and height) is accessible through the `Width` and `Height` properties of the `Bitmap` object. The image is raster-based, and these values are given in pixels.

The image is also stored at a certain resolution; that is, it has an associated DPI value. In fact, the image must have a certain number of dots in the horizontal direction and a certain number of dots in the vertical direction. Hence, the `Bitmap` class makes these values available through the `HorizontalResolution` and `VerticalResolution` properties. If the pixels are square (which is usually the case), these two values are the same, and we just talk about the image's resolution.

Furthermore, the drawing surface (in our case, the screen) also has a resolution. As we've discussed in earlier chapters, the image resolution of a screen is assumed to be 96 DPI. Whatever the drawing surface, you can obtain the horizontal and vertical resolution of the drawing surface through the `DpiX` and `DpiY` properties of the `Graphics` class.

GDI+ automatically calculates the display size of the image on your screen based on the size of the image, the resolution of the image, and the resolution of the screen. The default behavior is to use the following formulae:

Display Image Width = Actual Image Width X Display Resolution/Image Resolution

Display Image Height = Actual Image Height X Display Resolution/Image Resolution

These equations are derived from the fact that, by default, the ratio between the display image size and the actual image size is equal to the ratio between the display resolution and the image resolution. In other words:

Display Image Width/Actual Image Width = Display Resolution/Image Resolution

Display Image Height/Actual Image Height = Display Resolution/Image Resolution

To see this at work, let's add some code to our example. The code in boldface tells you the screen resolution and image resolution, and the width and height of the image contained in memory in the `Bitmap` object. Then we use a `Size` object to calculate the display width and height of the image on the screen.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.DrawImage(bmp, 0, 0);

    Console.WriteLine("Screen resolution: " + g.DpiX + "DPI");
    Console.WriteLine("Image resolution: " +
        bmp.HorizontalResolution + "DPI");
```

```

Console.WriteLine("Image Width: " + bmp.Width);
Console.WriteLine("Image Height: " + bmp.Height);

SizeF s = new SizeF(bmp.Width*(g.DpiX/bmp.HorizontalResolution),
                    bmp.Height*(g.DpiY/bmp.VerticalResolution));
Console.WriteLine("Display size of image: " + s);
}

```

When you run this code, it will display the image and also write the output to the console window:

```

Screen resolution: 96DPI
Image resolution: 300DPI
Image Width: 1000
Image Height: 1100
Display size of image: {Width=320, Height=352}

```

Of course, the exact output will depend on the size and resolution of the image you're using. This output tells us that the image of Rama the dog is 300 DPI and 1000 X 1100 pixels. The main point of this exercise is to show that, by default, GDI+ uses the ratio between the display resolution and the screen resolution to work out exactly how to display the image on the screen.

Controlling the Scaling of an Image

Unplanned automatic scaling, in the manner described in the [previous section](#), incurs a performance penalty, so you may want to avoid it. One alternative is to create a resized copy of the bitmap ahead of time—either in memory or on the disk—so that you can then draw it without scaling. This technique doesn't alter the final appearance of the bitmap; it simply improves the performance profile. If you have an application that is written in such a way that it often does unplanned automatic scaling, it is possible that restructuring your code will eliminate the performance penalty. As an example, if you have a window that contains many thumbnails, and each of these thumbnails is drawn with unplanned automatic scaling, you may see a significant performance hit when the window for this application is exposed. Keeping resized thumbnails at hand will solve the problem.

A real-life situation is one where you have a window in which you want to display a number of images of various sizes. The image files that are the source for your window might also be of various sizes. This doesn't present a problem, though. You can control the scaling explicitly when you display an image. The `DrawImage` method has 16 overloads, and some of these overloaded methods allow you to specify both a source rectangle and a destination rectangle.

- The *source rectangle* refers to the image, and it allows you to specify a subset of the source image to draw. If you specify a rectangle that is the same size as the image itself, `DrawImage` will draw the entire image.
- The *destination rectangle* refers to the drawing surface. If you specify that the destination rectangle should be the exactly same size (in pixels) as the source rectangle, `DrawImage` will draw the image without any resizing at all. If you specify a destination rectangle that is smaller or larger than the image size, `DrawImage` will perform the appropriate scaling (to make the source image fit the destination rectangle), before it draws the image. If you do not specify the destination rectangle, GDI+ will compare the resolution of the image and the resolution of the drawing surface, and decide whether or not to scale the image for you.

Displaying an Image without Scaling

In the following example, GDI+ draws an image, pixel for pixel, to the form. We create a rectangle whose dimensions are the same as the dimensions of the image, and then use that rectangle as both the source and destination rectangle. Because the dimensions of the destination rectangle are the same as those of the image itself, this method disregards the resolution of the image and the window, and draws the image verbatim:

```

private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");

    Rectangle r = new Rectangle(0, 0, bmp.Width, bmp.Height);
    g.DrawImage(bmp, r, r, GraphicsUnit.Pixel);
}

```

Scaling the Image to Fit the ClientRectangle

Another common operation is to draw an image so that it takes up the client area of a form or a custom control. You can use another overload of the `DrawImage` method to do this:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");

    Rectangle r = new Rectangle(0, 0, bmp.Width, bmp.Height);
    g.DrawImage(bmp, this.ClientRectangle);
}
```

Now, when you resize the window, you see your image resized to fill the client area of the window, as shown in [Figure 5-2](#).



Figure 5-2: Stretched bitmaps

Note No animals were harmed during the production of this book.

Changing the Resolution of a Bitmap

You can change the resolution of a bitmap by calling the `SetResolution` method of the `Bitmap` object. When you change the resolution of the bitmap, it affects how GDI+ resizes the bitmap when it draws it. To demonstrate this, modify the `Paint` event from the previous example as follows:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)

{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");

    g.FillRectangle(Brushes.White, this.ClientRectangle);
    bmp.SetResolution(600f, 600f);
    g.DrawImage(bmp, 0, 0);
    bmp.SetResolution(1200f, 1200f);
    g.DrawImage(bmp, 180, 0);
}
```

When this code is run, you see the results shown in [Figure 5-3](#).

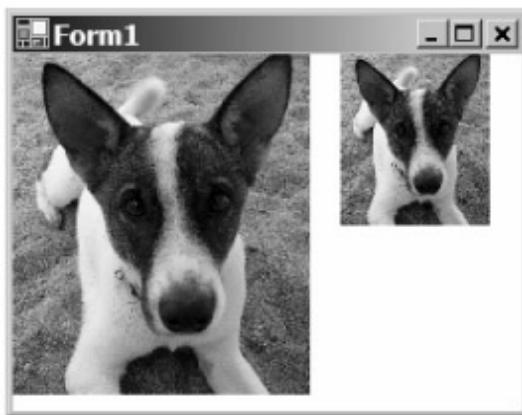


Figure 5-3: Effect of changing resolution

The resolution of the original image of Rama the dog (`rama.jpg`) is 300 DPI, and the image is 1000 X 1100 pixels. Hence, the width of the image in memory is $1000/300 = 3.333$ inches. The height in memory is $1100/300 = 3.667$ inches.

We've obtained the image on the left by changing the resolution of *the image in memory* to 600 DPI, and then displaying the result on the drawing surface. The change of resolution has the effect of changing the "physical dimensions" of the bitmap in memory. The width is $1000/600 = 1.667$ inches, and the height is $1100/600 = 1.833$ inches.

The image on the right is obtained in the same way. This time, we changed the resolution of the image in memory to 1200 DPI, which has the effect of changing the physical dimensions of the bitmap in memory to a width of $1000/1200 = 0.833$ inch and a height of $1100/1200 = 0.916$ inch.

The point here is that GDI+ pays attention to the resolution of the bitmap, and if you don't do any explicit scaling, GDI+ will use the resolution of the bitmap to scale the image.

Resizing and Interpolation

As you've seen, sometimes you scale an image explicitly, and sometimes GDI+ implicitly scales the image as a consequence of the drawing operations you've requested. Either way, whenever an image is resized, GDI+ uses one of several algorithms to do the resizing. When GDI+ needs to resize an image to produce a *larger* image (with a *greater* number of pixels), it uses an image manipulation algorithm to determine the color of the new pixels. This process is called *interpolation*.

Interpolation is the process of using the known values of two points to estimate or calculate the values at other points that lie between. A demographer who knows the population of a town in the year 1960 and in the year 1965 can use those values to reasonably estimate what the town's population was in, say, 1962. When we increase the resolution of an image, some of the pixel points in the new image correspond to pixel points in the old image, and so we can consider that we know the colors of those pixel points. But the new image, being bigger, has *new* pixel points whose color we don't know. So, GDI+ uses the color values of nearby pixels and applies similar interpolation techniques to determine the colors of those new pixels.

When resizing an image, and particularly when enlarging one, there is always a trade-off between processing time and image quality. An algorithm that accurately interpolates the values of new pixels will create a higher-quality image but will take longer. You might not need to worry at all about using CPU time when using Windows Forms and resizing a few images. However, when using GDI+ to generate images dynamically on a busy web server, you need to consider the trade-off between processing time and image quality.

For this reason, GDI+ allows you to control the image resizing quality programmatically, through the value you assign to the `Graphics.InterpolationMode` property. This property takes a value from the `InterpolationMode` enumeration (which belongs to the `System.Drawing.Drawing2D` namespace). This enumeration has five values, which denote a specific level of quality. In ascending order of quality, they are as follows:

- `NearestNeighbor`

- Bilinear (the default)
- HighQualityBilinear
- Bicubic
- HighQualityBicubic

In addition, you can use three more values in the `InterpolationMode` enumeration:

- Default
- High
- Low

To see the `Graphics.InterpolationMode` property in action, let's modify our previous example as follows:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    int width = bmp.Width;
    int height = bmp.Height;

    // Resize the image using the lowest quality interpolation mode

    g.InterpolationMode = InterpolationMode.NearestNeighbor;
    g.DrawImage(
        bmp,
        new Rectangle(10, 10, 120, 120),      // source rectangle
        new Rectangle(0, 0, width, height), // destination rectangle
        GraphicsUnit.Pixel);

    // Resize the image using the highest quality interpolation mode
    g.InterpolationMode = InterpolationMode.HighQualityBicubic;
    g.DrawImage(
        bmp,
        new Rectangle(130, 10, 120, 120), // source rectangle
        new Rectangle(0, 0, width, height), // destination rectangle
        GraphicsUnit.Pixel);
}
```

When you run the example, you should see something like [Figure 5-4](#). The difference in quality may not be too clear in this figure, but you should be able to see on your screen that the image on the right is of far higher quality.

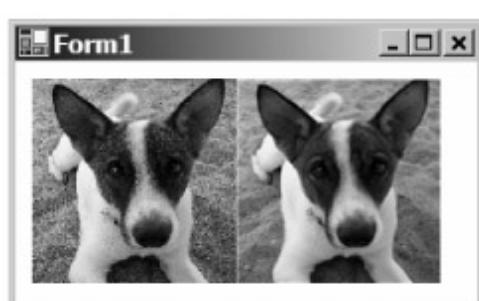


Figure 5-4: Different quality images

[Previous](#)[Next](#)

Manipulating Images

When you use GDI+'s `Bitmap` object to represent a bitmap image stored in memory, you can perform plenty of operations to manipulate that bitmap. For example, you can crop it, skew it, reflect it, and clone it. You can even retrieve thumbnail information from it. You'll see examples of all these in this section.

Cropping an Image

Cropping an image is a common tool in image manipulation. Quite simply, *cropping* involves selecting a rectangular portion of the image and trimming (and throwing away) the parts of the image that lie outside that rectangle. The size of the resulting, cropped image is, naturally, smaller than the original—it's the same size as the rectangle you used to define the crop.

The following code draws a cropped image:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)

{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Rectangle sr = new Rectangle(80, 60, 400, 400);
    Rectangle dr = new Rectangle(0, 0, 200, 200);
    g.DrawImage(bmp, dr, sr, GraphicsUnit.Pixel);
}
```

The source rectangle, `sr`, is a rectangle whose top-left corner is at location (80, 60) and whose dimensions are 400X400 pixels. This rectangle defines the portion of the original image that we want to keep after we've cropped it. As you can see in [Figure 5-5](#), it contains Rama's right ear. The destination rectangle, `dr`, describes the location within the drawing surface where we want to draw the result and the size we want it to be.



Figure 5-5: Cropped image

Skewing, Reflecting, and Rotating an Image

One of the overloaded `DrawImage` methods allows you to pass an array of three points as an argument. These three points perform the following functions when the image is drawn:

- The first point specifies the destination of the upper-left point of the original bitmap.
- The second point specifies the destination of the upper-right point of the original.

- The third point specifies the destination of the lower-left point of the original.

You can *scale* or *skew* the image, *reflect* it either horizontally or vertically, or *rotate* it, simply by choosing the appropriate values for these three destination points. In fact, if you wish, you can achieve any combination of these manipulations on an image simultaneously, with just a single operation based on the values of your three points. For simplicity, we'll deal with them one at a time in the examples here.

Skewing an Image

First, let's try skewing our image of Rama the dog. Here's some code that will do the job:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Point[] destinationPoints = {
        new Point(0, 0), // destination for upper-left point of original
        new Point(100, 0), // destination for upper-right point of original
        new Point(50, 100)}; // destination for lower-left point of original
    g.DrawImage(bmp, destinationPoints);
}
```

When run, it looks like [Figure 5-6](#). As you can see, we've set the destination points of three of the corners of the image in such a way that GDI+ must skew the image into a parallelogram shape.



Figure 5-6: Skewed image

Reflecting an Image

Now, let's modify the array of points in order to create the effect of a reflection in a horizontal mirror. To do this, we tell GDI+ that the destination of the top-left corner is *below* the destination of the bottom-left corner on the drawing surface:

```
Point[] destinationPoints = {
    new Point(0, 100), // destination for upper-left point of original
    new Point(100, 100), // destination for upper-right point of original
    new Point(0, 0)}; // destination for lower-left point of original
g.DrawImage(bmp, destinationPoints);
```

The result is that we see Rama's reflection in a puddle, as shown in [Figure 5-7](#).



Figure 5-7: Reflected image

Rotating an Image

To rotate the image 90 degrees to the right, set up the array as follows:

```
Point[] destinationPoints = {
    new Point(100, 0), // destination for upper-left point of original
    new Point(100, 100), // destination for upper-right point of original
    new Point(0, 0)}; // destination for lower-left point of original
g.DrawImage(bmp, destinationPoints);
```

This simply places the upper-left corner of the source in the upper-right corner of the destination, the lower-left corner of the source in the upper-left corner of the destination, and so on. [Figure 5-8](#) shows the rotated image.



Figure 5-8: Rotated image

The result is similar to the effect you get when you use the `Bitmap.RotateFlip` method. However, here, you are using the `DrawImage` method to manipulate the image *as you're drawing it to the drawing surface*, so it *doesn't* affect the image stored in memory. In contrast, the `Bitmap.RotateFlip` method is a manipulation of the image stored in memory itself.

Cloning an Image

The abstract `Image` class provides a `Clone` method, which allows you to make a copy of an existing `Image`, `Metafile`, or `Bitmap` object. In fact, the `Clone` method is overloaded, so you can use it in a number of ways. We will not cover them all here. They're all listed in the GDI+ documentation at <http://msdn.microsoft.com/library>.

Let's examine one example. The `Clone` method has an overload that allows you to specify a source rectangle with which you can specify the portion of the source image. In this case, the `Clone` method produces a clone of the specified portion.

The following code demonstrates how to clone a portion of an existing `Bitmap` object to create a new `Bitmap` object:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Rectangle r = new Rectangle(120, 120, 400, 400);
    Bitmap bmp2 = bmp.Clone(r, System.Drawing.Imaging.PixelFormat.DontCare);
    g.DrawImage(bmp, new Rectangle(0, 0, 200, 200));
    g.DrawImage(bmp2, new Rectangle(210, 0, 200, 200));
    bmp2.Dispose();
}
```

Caution When you specify the source rectangle, you need to make sure that the rectangle is within the bounds of the source image. Otherwise, you will get a `System.OutOfMemoryException` exception.

When you run this example, you see the two images, as shown in [Figure 5-9](#). The image on the left is a display of the original; the one on the right is a display of the clone.



Figure 5-9: Cloned image

Getting a Thumbnail of an Image

Some image formats have the capability to contain more than just the image itself. The image source might contain information about multiple images, and in particular, a thumbnail image. The `Image` class provides a method called `GetThumbnailImage`, which allows you to get a thumbnail of the image. If the image source contains thumbnail data, this method gets it. If not, then the method scales the image to create a thumbnail image.

If the source image contains the thumbnail, and the thumbnail doesn't need to be created when you request it, you will get a performance benefit. If you were creating a image catalog program that allowed the user to browse by looking at thumbnails, you would probably want to make sure that the thumbnails were stored in the images to eliminate unnecessary processing when moving from one group of images to another.

Before demonstrating this method, I need to make a brief point about the arguments that the method expects. Here is the signature of the `GetThumbnailImage` method:

```
public Image GetThumbnailImage(
    int thumbWidth,
    int thumbHeight,
    Image.GetThumbnailImageAbort callback,
    IntPtr callbackData
);
```

The last two arguments of this method (`callback` and `callbackData`) are not used in GDI+ version 1.0. Even though they are not used, they're not optional, so you still need to declare a delegate for the `callback` method that does nothing other than return `false`, and pass the delegate to this method.

Therefore, we first declare our `ThumbnailCallback` method, as follows:

```
public bool ThumbnailCallback()
{
    return false;
}
```

Now, we're in a position to write a brief example within our `Paint` event handler:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    Image.GetThumbnailImageAbort thumbnailCallback =
        new Image.GetThumbnailImageAbort(ThumbnailCallback);
    Image tn = bmp.GetThumbnailImage(
        40, 40, thumbnailCallback, IntPtr.Zero);
    g.DrawImage(tn, 0, 0, tn.Width, tn.Height);
    tn.Dispose();
}
```

}

In this code, we declare the delegate as follows:

```
Image.GetThumbnailImageAbort thumbnailCallback =
    new Image.GetThumbnailImageAbort(ThumbnailCallback);
```

Then we use this delegate when we call the `GetThumbnailImage` method. After getting the thumbnail image, we draw it, as shown in [Figure 5-10](#).



Figure 5-10: Thumbnail image

[Previous](#)

[Next](#)

[Previous](#)

[Next](#)

Creating and Drawing into an Image

Of course, you won't always want to work with images loaded from disk. Sometimes you'll want to draw your own images. In that case, you need to create a brand-new, empty image object, and then perform a number of operations to build up the elements of your new image in memory. You may even want to combine images to create different and exciting effects. In this section, you'll see how GDI+ allows you to do all these things.

Creating a New Bitmap

When you want to create a new bitmap object into which to draw, you can use one of the `Bitmap` class constructors. Although a variety of options available, we are interested in three constructors here:

```
Bitmap bmp = new Bitmap(width, height);
Bitmap bmp = new Bitmap(width, height, pixelFormat);
Bitmap bmp = new Bitmap(width, height, g);
```

In all of these constructors, `width` and `height` indicate the width and height of the new image in pixels, and they must be integers. These constructors work as follows:

- If you use the first constructor, you get a new `Bitmap` object that represents a bitmap in memory with a `PixelFormat` of `Format32bppARGB`.
- If you use the second constructor, you can specify your own choice of `PixelFormat`. Here, `pixelFormat` is a value that is taken from the `PixelFormat` enumeration, which we discussed earlier in this chapter.
- If you use the third constructor, you get a new `Bitmap` object representing a bitmap in memory whose resolution and color depth are the same as those of the specified drawing surface (here, `g` is the `Graphics` object, which represents the drawing surface).

Once you've created the `Bitmap` object, you can get an instance of the `Graphics` class that encapsulates a drawing surface for the bitmap. Using this `Graphics` object, you can draw into the bitmap. Then, during the `Paint` event, you can use the `Graphics` object that represents the screen drawing surface to draw the image to the screen.

To make this clear, we can put it all into the `Paint` method handler, like this:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Bitmap bmp = new Bitmap(100, 100);
```

```
Graphics gImage = Graphics.FromImage(bmp);
gImage.FillRectangle(Brushes.Red, 0, 0, bmp.Width, bmp.Height);
gImage.DrawRectangle(Pens.Black, 10, 10, bmp.Width-20, bmp.Height-20);

Graphics gScreen = e.Graphics;
gScreen.FillRectangle(Brushes.White, this.ClientRectangle);
gScreen.DrawImage(bmp, new Rectangle(10, 10, bmp.Width, bmp.Height));
}
```

Note in particular that we use *two* instances of the `Graphics` class here. The first, `gImage`, represents the drawing surface of the image. The second, `gScreen`, is more familiar; it represents the drawing surface of the form, which is the final destination of our image. [Figure 5-11](#) shows the result.

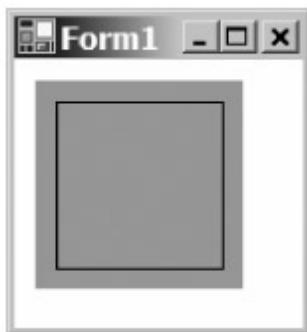


Figure 5-11: Drawing into an image

A Note on Scope

In the previous example, we put all the code in the `Paint` event handler for simplicity. In a more realistic application, it often makes more sense to give the `Bitmap` object greater scope, because you will probably want to use it in a number of methods of the containing class. You might create the `Bitmap` object at the `form` level, like this:

```
public class Form1 : System.Windows.Forms.Form
{
    private Bitmap bmp;
    ...
}
```

Then, if you wanted to create and draw the new image at the time you load the control, you would use the `Load` event handler to do it, like this:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    bmp = new Bitmap(100, 100);
    Graphics g = Graphics.FromImage(bmp);
    g.FillRectangle(Brushes.Red, 0, 0, bmp.Width, bmp.Height);
    g.DrawRectangle(Pens.Black, 10, 10, bmp.Width-20, bmp.Height-20);
}
```

Finally, you would paint the resulting bitmap to the drawing surface, through the `Paint` event handler:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    g.DrawImage(bmp, new Rectangle(10, 10, bmp.Width, bmp.Height));
}
```

Advantages of Drawing to a Bitmap Object

In some cases, you gain a number of benefits when you choose to create and draw into a bitmap, and then draw the bitmap to the form (instead of drawing directly to the form):

- **Performance:** You may wish to draw into an image ahead of time, and then draw the image in the Paint event when you need it. This means you need to draw the graphics one time only, instead of drawing them each time that you receive a Paint event. If the drawing is time-intensive, this gives you a definite performance improvement.
- **Dynamic image generation:** Sometimes you want to include a dynamically generated graphic in a web page. To do this, you draw into an image on the web server, and then include a reference to the dynamically generated image in your web page.
- **Complex three-dimensional graphics preparation:** If your graphic has multiple layers, where you first need to draw the background, then mid-ground, and then foreground, drawing into an image off-screen allows you to prepare the image completely in advance of the time you render to the screen. This prevents the unsightly flashing effect that comes from painting one layer over another on the screen. Moreover, some image processes, such as blurring and sharpening, can be achieved only by manipulating the image directly before it's rendered. When you draw the image in advance, those facilities are available to you.

Double-Buffering

If you need to create and present complex, layered graphics, you surely must be concerned about the performance of your code. To create such graphics in a more efficient manner, you can use a technique called *double-buffering*.

In short, double-buffering is the process of drawing to an image, and then drawing the completed image to a form. Effectively, you use the image drawing surface as an off-screen buffer, which allows you to prepare the image completely before you paint it to the screen.

Double-buffering is particularly useful when you're drawing graphics that involve *layering* of many different elements, one on top of another, at the same location in the drawing surface, to build up a three-dimensional effect. The problem is that creating such effects within a single image tends to take up a significant amount of processing time. Double-buffering allows you to prepare your graphic ahead of time, and then draw it to the screen in a single operation. This will create a much smoother appearance to the user.

For example, consider a situation where you need to draw a complex map with many elements: rural and urban backgrounds, highways, rivers, symbols, road names, and area names. You want to avoid the job of drawing each of these elements individually onto a single image, because a great deal of extra effort is involved in getting the relative levels of each element exactly right, and it's also a lot of work for the processor. On a slower processor, you could see these graphical operations being performed individually, one after another. This may cause a flickering effect because the user sees the graphic for a very small amount of time after the background is drawn but before the foreground is drawn. A solution is to use double-buffering to build up the image in layers. You draw the background first; then you add geographical regions in a different color on top of the background; and then you draw rivers, cities, highways, railroad tracks, and other elements on top of the geographical regions. Finally, you might add text labels to many of these points. When everything is in place, you paint the whole lot to the screen in one go.

Windows Forms and GDI+ support double-buffering seamlessly. You can set three flags, and then GDI+ will take care of all details of the double-buffering for you. To enable doublebuffering, you need to use the `SetStyle` method of the `Form` class. Placed in either the constructor for the form or the `Load` event for the form, the following code enables double-buffering:

```
SetStyle(ControlStyles.DoubleBuffer, true);
SetStyle(ControlStyles.AllPaintingInWmPaint, true);
SetStyle(ControlStyles.UserPaint, true);
```

The definitions of the members of the `ControlStyles` enumeration are as follows:

- `DoubleBuffer`: Indicates that all drawing operations are to be performed in a buffer (an image), and that after drawing operations are complete, the result is drawn to the screen.
- `AllPaintingInWmPaint`: Indicates that the background of the window should not be drawn in a separate operation. You should set this style only if you set `UserPaint` to `true`.
- `UserPaint`: Indicates that the control (rather than the WindowsForms framework) should paint itself.

You set `AllPaintingInWmPaint` and `UserPaint` to `true` when your control draws its entire surface. The .NET Framework doesn't draw any part of the control, including the background of the control.

With these styles set, when GDI+ raises the `Paint` event, it refrains from setting the drawing surface to be the window. Instead, it creates an image in memory and a `Graphics` object that allows you to use that image as a drawing surface. Then it passes that `Graphics` object in as part of the argument to the `Paint` event. This all happens behind the scenes. Your drawing code doesn't need to know that it is drawing to an image rather than directly to the screen. After the `Paint` event returns, the .NET Framework then draws the prepared image to the form or window.

In today's world of fast processors and 3D graphics accelerators, it is a little difficult to come up with an artificial example that demonstrates the benefits of double-buffering. After all, if CPUs and graphics accelerators are capable of running Doom III, you would need to throw a lot of graphics operations at them before they would even flinch! For this reason, I've elected not to include a double-buffering example. However, the example in the [previous section](#) demonstrates the *principle* of double-buffering: we prepared the red rectangle off screen in a `Bitmap` object, and we displayed it to the screen only after we completely finished it.

Working with Alpha in Images

As explained in [Chapter 2](#), you can use the alpha component to control the level of transparency of a color. You saw how to create a custom `Color` object composed of alpha, red, green, and blue components (and hence has a color and a level of transparency), and then assign the `Color` to a brush or pen. You could then use it to draw lines, arcs, rectangles, text, and so on, with a semitransparent effect.

Whether the drawing surface is a form or an image in memory, these facilities work in much the same way. You draw elements to the drawing surface, and as a result, the individual pixels of the drawing surface are assigned a color and a level of transparency. In addition, you can use alpha in images to implement further special effects. For example, you can make an entire image semitransparent, and when you draw new elements into an existing image, you can draw semitransparently, so that your drawing operations are blended with the existing contents of the image.

When you draw with a semitransparent color (a color that contains alpha information) into an image that already contains alpha information, you need to think about how GDI+ calculates the color and alpha of the individual pixels in the resulting image. For each affected pixel, GDI+ needs to combine the existing color and alpha with the color and alpha of your drawing color, to produce a resultant color and alpha that have the effect you desire.

GDI+ will calculate the new colors of the affected pixels in one of two ways:

- The drawing color and alpha may be *blended* with the existing color and alpha, to create a result that is dependent on the colors and alphas of both.
- The drawing color and alpha may simply *overwrite* the existing color and alpha in the pixel.

You can control whether a drawing operation *blends* with the image or *overwrites* its alpha information onto the image through the `CompositingMode` property of the `Graphics` class. This property has two possible values, which belong to the `CompositingMode` enumeration (part of the `System.Drawing.Drawing2D` namespace):

- `SourceOver`: The drawing operation is blended with the existing pixels of the image. The blend is determined by the alpha component of the color being drawn. This is the default behavior.
- `SourceCopy`: The drawing color and alpha overwrite the existing bits in the image.

We can build a small example to demonstrate this graphically. First, let's draw some vertical bars in a window:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics gForm = e.Graphics;
    gForm.FillRectangle(Brushes.White, this.ClientRectangle);
    for (int i = 1; i <= 7; ++i)
    {
        int width = 100 / 7;
        int height = 100 * i;
        int x = (i - 1) * width;
        int y = 100 * i;
        gForm.FillRectangle(Brushes.Blue, x, y, width, height);
    }
}
```

```

    Rectangle r = new Rectangle(i*40-15, 0, 15,
        this.ClientRectangle.Height);
    gForm.FillRectangle(Brushes.Orange, r);
}
}

```

When you run the example without drawing any images with alpha in them, the bars are drawn to the window as shown in [Figure 5-12](#).

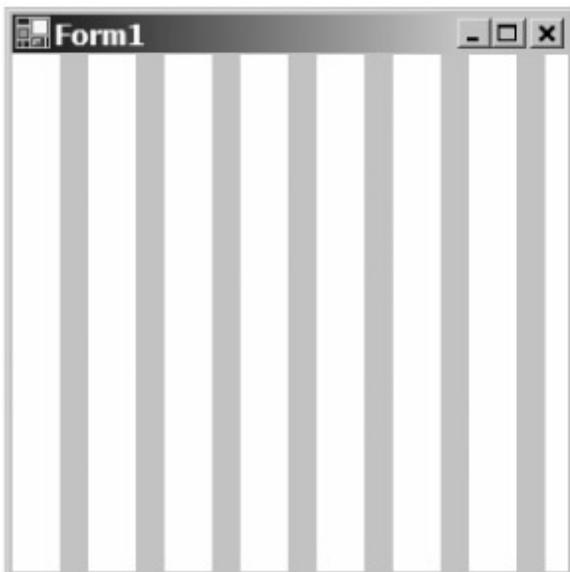


Figure 5-12: Bars drawn in a window

Now, we'll create a bitmap in memory, and draw it over the vertical bars. To do that, we'll modify the Paint event as follows:

```

private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics gForm = e.Graphics;
    gForm.FillRectangle(Brushes.White, this.ClientRectangle);
    for (int i = 1; i <= 7; ++i)
    {
        Rectangle r = new Rectangle(i*40-15, 0, 15,
            this.ClientRectangle.Height);
        gForm.FillRectangle(Brushes.Orange, r);
    }

    // Create a bitmap image in memory and set its CompositingMode
    Bitmap bmp = new Bitmap(260, 260,
        System.Drawing.Imaging.PixelFormat.Format32bppArgb);
    Graphics gBmp = Graphics.FromImage(bmp);
    gBmp.CompositingMode =
        System.Drawing.Drawing2D.CompositingMode.SourceCopy;

    // Create a red color with an alpha component
    // then draw a red circle to the bitmap in memory
    Color red = Color.FromArgb(0x60, 0xff, 0, 0);
    Brush redBrush = new SolidBrush(red);
    gBmp.FillEllipse(redBrush, 70, 70, 160, 160);

    // Create a green color with an alpha component
    // then draw a green rectangle to the bitmap in memory
    Color green = Color.FromArgb(0x40, 0, 0xff, 0);
    Brush greenBrush = new SolidBrush(green);

    gBmp.FillRectangle(greenBrush, 10, 10, 140, 140);
}

```

```
// Now draw the bitmap on our window
gForm.DrawImage(bmp, 20, 20, bmp.Width, bmp.Height);

// Dispose of all objects that consume resources
bmp.Dispose();
gBmp.Dispose();
redBrush.Dispose();
greenBrush.Dispose();
}
```

Note the following about this example:

- Two `Graphics` objects are here: `gForm` represents the form drawing surface, and `gBmp` represents the bitmap in memory.
- We've set the `CompositingMode` property of the bitmap to `SourceCopy`, so we should expect that the green square (drawn second) should *overwrite* the red circle within the bitmap.
- We have not specified a value for the `CompositingMode` property of the form drawing surface, `gForm`. Therefore, it will take the default value, which is `SourceOver`, and we should expect the circle and square to *blend* with the bars when we draw the bitmap image over the orange bars.

To confirm our expectations, let's run the example. The result is shown in [Figure 5-13](#). As expected, the square has *overwritten* the circle at the pixels where they overlap, and the square and circle combination has *blended* with the bars.

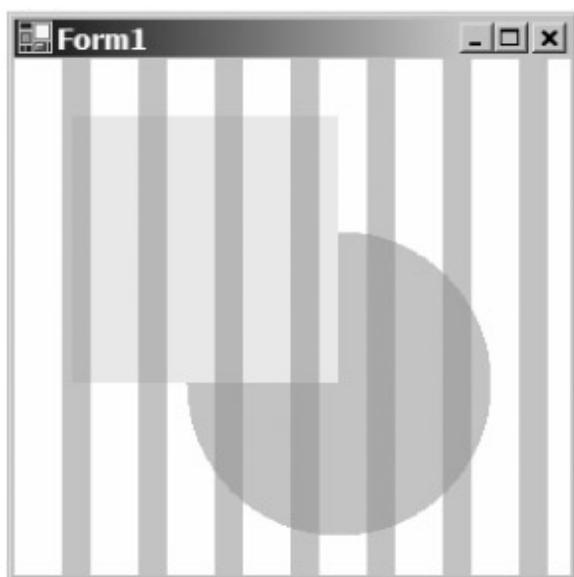


Figure 5-13: Alpha blending in action

Getting and Setting Pixels

In previous chapters, we've talked about the importance of having control over the color and alpha of individual pixels on the drawing surface. For example, when creating 3D effects, just one pixel out of place can spoil the effect.

In GDI+, the `Bitmap` class provides methods for retrieving and setting the color (including the alpha value) of individual pixels. The following example demonstrates the technique, by creating a small image, drawing into the image, and then printing out the color values of each pixel in the image:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics gForm = e.Graphics;
    gForm.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a bitmap in memory
```

```

Bitmap bmp = new Bitmap(6, 6);
Graphics gBmp = Graphics.FromImage(bmp);

// Give the bitmap a white background
gBmp.FillRectangle(Brushes.White, 0, 0, bmp.Width, bmp.Height);

// Draw a red diagonal line in the bitmap
gBmp.DrawLine(Pens.Red, 0, 0, 5, 5);

// Now draw the bitmap on our window
gForm.DrawImage(bmp, 20, 20, bmp.Width, bmp.Height);

// Finally, write the pixel information to the console window
for (int y = 0; y < bmp.Height; ++y)
{
    for (int x = 0; x < bmp.Width; ++x)
    {
        Color c = bmp.GetPixel(x, y);
        Console.Write("{0,2:x}{1,2:x}{2,2:x}{3,2:x} ",
                      c.A, c.R, c.G, c.B);
    }
    Console.WriteLine();
}
bmp.Dispose();
}

```

When highly magnified, the image looks like [Figure 5-14](#).

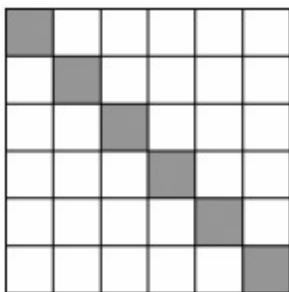


Figure 5-14: A diagonal line

The output to the console is as follows:

```

ffff 0 0 ffffffff ffffffff ffffffff ffffffff ffffffff
fffffff ffff 0 0 ffffffff ffffffff ffffffff ffffffff
fffffff ffffffff ffff 0 0 ffffffff ffffffff ffffffff
fffffff ffffffff ffffffff ffff 0 0 ffffffff ffffffff
fffffff ffffffff ffffffff ffff 0 0 ffffffff ffffffff
fffffff ffffffff ffffffff ffffffff ffff 0 0 ffffffff

```

This output consists of the alpha—red—green—blue hexadecimal values for each of the 36 pixels in the image. These values are returned by the `Bitmap` object's `GetPixel` method. You can see that most of the pixels are pure white—they have a value of `fffffff` (alpha=255, R=255, G=255, B=255). The pixels in the diagonal line are different—they have the value `fff 0 0` (alpha=255, R=255, G=0, B=0), and these pixels are red. You can see the direct relationship between the bitmap in [Figure 5-14](#) and the hexadecimal output.

Now try adding the following lines, to change the value of some of these pixels to green:

```

// Create a green color and use it to change some pixels
Color green = Color.FromArgb(0x40, 0, 0xff, 0);
for (int x = 0; x < bmp.Height; ++x)
{
    bmp.SetPixel(x, x, green);
}

```

PIXEL CONTROL FOR DIGITAL IMAGE MANIPULATION

Pixel control is also useful when performing certain kinds of digital-image manipulation. Consider, for example, a project I undertook recently, in which I needed to remove graininess from very old black-and-white photographs, without excessively blurring the images. I used the following technique:

1. Scan the negative with a very high-quality scanner, good enough so that I could see the individual grains of the negative when looking at the image at high magnification.
2. Apply an algorithm that locates the position and size of each photographic grain of the negative.
3. After locating adjacent grains, calculate a geometric description of the boundaries between the grains.
4. Average the brightness of the pixels within one geometric shape that contains one photographic grain.

In the later stages of this process, control of individual pixels was an important factor.

Creating Semitransparent Images by Applying Color Transformations

GDI+ supports facilities for applying *color transformations* (including the alpha component of colors). Color transformations allow you to create some impressive effects. For example, you might want to make the red component of all colors of an image brighter, or make the image transparent wherever there is blue in the image.

Note Making an image transparent wherever there is blue is similar to a technique that TV weather forecasters have been using for years. We see them waving their hands over the map, showing the locations of highs, lows, and so on, but they are actually standing in front of a blue wall in the TV studio, while a visual effect superimposes their image onto the computer-generated weather chart. In fact, these days many apparently elaborate TV news studios exist only *virtually*. The news anchors sit at a desk in a blue room, and the designer uses software to create the illusion of an elaborately decorated, busy newsroom with computer monitors and people working in the background. The software can even simulate the optical depth of field of the TV camera, so that items and people in the background are the appropriate level of blurriness. This makes it very easy to redecorate the newsroom!

Building custom controls doesn't typically involve color transformations. However, one operation that is useful is that of making an image semitransparent. To achieve this effect, you use the `ImageAttributes` class, which allows you to modify the functionality of GDI+ and change how colors in an image are manipulated during rendering.

You also use the `ColorMatrix` class here. This class uses *matrix algebra* to help achieve the desired effect. GDI+ represents the color of each pixel in a bitmap as a color vector with four elements (alpha, red, green, and blue), and it happens that we could use a 4 X 4 matrix of numeric entries to apply *linear transformations* (such as color rotation and color scaling) to an image. However, the 4 X 4 matrix would not give us facilities for applying color translations. If we add a "dummy" fifth column and row, we can use the fifth column for translation after doing a linear transformation.

The simplest color matrix is the *identity matrix*. If you perform a transformation by applying the identity matrix, the result is the same image you started with. In other words, the identity matrix doesn't change anything. For mathematical reasons that we won't go into here, an identity matrix is always a square matrix with ones down the leading diagonal and zeros elsewhere, as shown in [Figure 5-15](#). A good approach when trying out color transformations is to start with the identity matrix, and then make small adjustments until you get the transformation that you desire.

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Figure 5-15: The identity matrix

Performing a transformation to set the alpha component for an entire image is easy. If you leave all the other entries the same, you can change the entry in position [4][4] in the matrix to scale the alpha component of a color. Each entry in the matrix is a single-precision, floatingpoint value, so to set the alpha component to 60%, use the matrix shown in [Figure 5-16](#).

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	0.6	0
0	0	0	0	1

Figure 5-16: Alpha scaling matrix

Let's take a look at an example that uses this technique. The `ColorMatrix` and the `ImageAttributes` classes are in the `System.Drawing.Imaging` namespace, so include a `using` statement to that effect.

We'll draw a set of gray, vertical lines on the drawing surface, using code similar to something you saw earlier in this chapter:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Bitmap bmp = new Bitmap("rama.jpg");
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Draw a set of gray vertical bars on the form
    for (int i = 1; i <= 7; ++i)
    {
        Rectangle r = new Rectangle(i * 40, 15, 0, 15,
            this.ClientRectangle.Height);
        g.FillRectangle(Brushes.Gray, r);
    }
}
```

Next, we'll define our color matrix. It's the matrix shown in [Figure 5-16](#), which simply changes the alpha level to 60%:

```
// Create a color matrix
// The value 0.6 in row 4, column 4 specifies the alpha value
float[][] matrixItems = {
    new float[] {1, 0, 0, 0, 0},
    new float[] {0, 1, 0, 0, 0},
    new float[] {0, 0, 1, 0, 0},
    new float[] {0, 0, 0, 0.6f, 0},
    new float[] {0, 0, 0, 0, 1}};
ColorMatrix colorMatrix = new ColorMatrix(matrixItems);
```

Now we'll use an `ImageAttributes` object. We assign our color matrix as a property of the `ImageAttributes` object:

```
// Create an ImageAttributes object and set its color matrix
ImageAttributes imageAtt = new ImageAttributes();
```

```
    imageAtt.SetColorMatrix(  
        colorMatrix,  
  
        ColorMatrixFlag.Default,  
        ColorAdjustType.Bitmap);
```

Finally, we use the `ImageAttributes` object to draw the image of Rama the dog:

```
// Now draw the semitransparent bitmap image.  
g.DrawImage(  
    bmp,  
    this.ClientRectangle, // destination rectangle  
    0.0f,                // source rectangle x  
    0.0f,                // source rectangle y  
    bmp.Width,           // source rectangle width  
    bmp.Height,          // source rectangle height  
    GraphicsUnit.Pixel,  
    imageAtt);  
  
imageAtt.Dispose();  
}
```

In this example, the gray bars in the background help to show the transparency of the image. When you run the example, it appears as shown in [Figure 5-17](#).



Figure 5-17: Semitransparent image

Normally, you do this scaling when you draw an image to a drawing surface. However, by creating a new bitmap, and drawing the source bitmap into the new bitmap while doing the color or alpha scaling, you can also use the facility to transform an image without drawing it to a window or a printer.

◀ Previous

Next ▶

◀ Previous

Next ▶

Playing an Animation

We've talked about displaying static images in custom controls. GDI+ also provides a technique that allows you to display *animated images* in custom controls. Aside from the (debatable) niceties of having moving images, this has practical uses, too. For example, if your custom control needs to perform some operation that will take quite a bit of time, you might display a simple animation to let the user know that the operation is proceeding properly.

In this example, we demonstrate this tool using an animated GIF. This is simply a GIF file that contains a

number of images in a predefined format in such a way that they appear one after another, and thus give the effect of animation. Each image corresponds to a frame of the animation. Many freeware, shareware, and commercial tools allow you to create GIF files that contain animations. The `ImageAttribute` class contains the functionality that plays an animation from a GIF file.

Note For this example to work, the file that you load must be a GIF animation. A GIF animation called `arrow.gif` is provided along with the downloadable code for this book (available from the Downloads section of the Apress web site, www.apress.com). If you're running this in debug mode, place `arrow.gif` in the `/bin/debug` subdirectory of your application directory.

This example is slightly more complex than the others in this chapter. First, we need to create a Form-level variable to represent our animation:

```
public class Form1 : System.Windows.Forms.Form
{
    private Bitmap bmp;
    ...
}
```

Next, we need to add a Load event handler to the form. Modify it as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    bmp = new Bitmap("arrow.gif");
    ImageAnimator.Animate(bmp, new EventHandler(this.OnFrameChanged));
}
```

Note that if you do this manually, you'll need to add this line to `InitializeComponent`:

```
private void InitializeComponent()
{
    ...
    this.Load += new System.EventHandler(this.Form1_Load);
}
```

Next, we need a Paint event handler:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    //Get the next frame ready for rendering
    ImageAnimator.UpdateFrames();
    //Draw the next frame in the animation
    e.Graphics.DrawImage(this.bmp, new Point(0, 0));
}
```

Finally, add this private member method, which is used in the Paint event handler:

```
private void OnFrameChanged(object o, EventArgs e)
{
    // Invalidate the window to force a call to the Paint event handler.
    // If we had more items in the window than just the animation, we could
    // invalidate just the area occupied by the animation.
    this.Invalidate();
}
```

When you run the example, the animation will play, as shown in [Figure 5-18](#).

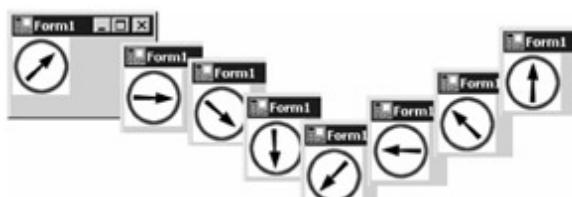


Figure 5-18: Playing an animation

Converting a BMP to a JPEG

Some file formats are more appropriate for certain uses than others. For example, the JPEG format is a good choice when compressing photographs. The GIF format is appropriate for computer-generated images that have rectangular regions that are a uniform color. The BMP format is used by some legacy software programs and may be used for compatibility. If you have a BMP file that contains a photograph, its footprint (the disk space it consumes) will be quite large in comparison to the same photograph stored as a JPEG. So, you may want to convert the BMP file to JPEG format.

GDI+ allows you to perform this kind of file conversion programmatically. If you want to do a simple BMP-to-JPEG file conversion, you can do it as follows:

```
Bitmap bmp = new Bitmap("GrapeBunch.bmp");
bmp.Save("GrapeBunch.jpg", System.Drawing.Imaging.ImageFormat.Jpeg);
```

This code assumes the default levels of compression and image quality in the resulting JPEG. With a little more code, you can control these factors within the application. This involves some extra work. Before you start, you need to be able to get the image encoder/decoder information for your desired target image type. To achieve this step, you can create a utility method, `GetEncoderInfo`, to get the encoder:

```
private static ImageCodecInfo GetEncoderInfo(String mimeType)
{
    int j;
    ImageCodecInfo[] encoders;
    encoders = ImageCodecInfo.GetImageEncoders();
    for(j = 0; j < encoders.Length; ++j)

        if(encoders[j].MimeType == mimeType)
            return encoders[j];
    return null;
}
```

This uses the `GetImageEncoders` method (which was introduced earlier in this chapter) to get the entire list of codecs. Then you just search through them looking for the right one.

With that information, you can start the conversion process. You need a `using` statement for the `System.Drawing.Imaging` namespace:

```
using System.Drawing.Imaging;
```

You use the utility method `GetEncoderInfo` to get the correct codec and assign it to the variable `ici`. Then declare a variable called `enc`, of type `Encoder`, and set it to `Encoder.Quality`. You also declare a variable called `eпа`, of type `EncoderParameter`, and initialize it with the desired level of compression:

```
ImageCodecInfo ici;
Encoder enc;
EncoderParameter ep;
EncoderParameters eпа;

// Initialize the necessary objects
ici = GetEncoderInfo("image/jpeg");
enc = Encoder.Quality;
eпа = new EncoderParameters(1);
```

When you perform the `Save` operation itself, you will need to pass an array of `EncoderParameter` objects. So, declare an array of length 1 and initialize the first element in the array to your initialized `EncoderParameter` object:

```
// Set the compression level
ep = new EncoderParameter(enc, 25L);
eпа.Param[0] = ep;
```

The second argument of the `EncoderParameter` constructor indicates the quality level. This argument must be a value between 0 and 100. The level of compression increases as the quality level reduces. This example specifies a 25% quality level.

Finally, you can load the BMP file and call the `Bitmap.Save` method, passing the `ImageCodecInfo` and the array of `EncoderParameter` objects as arguments:

```
// Create a Bitmap object from a BMP file  
Bitmap bmp = new Bitmap("GrapeBunch.bmp");  
  
// Convert to JPEG and save bmp.Save  
( "GrapeBunch.jpg", ici, epa );
```

This code provides an example of what you need to do if you are performing an advanced operation in GDI+ that requires a codec and encoder parameters. Some of the operations require more than one encoder parameter. In this case, you can construct the `EncoderParameters` passing the required number of encoder parameters and set the `Param` property as appropriate.

Previous

Next

Previous

Next

Summary

Whether you're writing an application that is image-driven or just putting together a simple application that would be enhanced by a few images, GDI+ gives you control over the use of images in your applications. When you're working with images, you're generally trying to do one of four things: loading or creating an image, displaying or rendering an image, manipulating an image, or saving the image. In this chapter, you saw how GDI+'s `Image` class, and the `Bitmap` and `Metafile` classes that inherit from it, provides you with this functionality.

You saw that GDI+ supports a number of different image formats. We looked at how these formats store the information (particularly in relation to the color depth, compression format, and pixel format). You have plenty of different formats to choose from, and your choice affects the quality of the stored or displayed image.

We focused on the `Bitmap` class, which is used to represent a raster-based image in memory, and is therefore of particular relevance. You saw how to use a `Bitmap` class constructor to load an image from a file on disk (such as a BMP, JPEG, GIF, or PNG file), and hence create a `Bitmap` object that represents a copy of the image in memory. You also saw how you can use other `Bitmap` class constructors to create a brand-new, empty image in memory, like a blank canvas onto which you can apply drawing operations and import portions of other images.

To display an image on your drawing surface, you can use the `Graphics.DrawImage` method. As you saw, it's worth thinking about checking the size and resolution of the image, and the resolution of the drawing surface, when deciding how big you want the image to appear on the drawing surface. It's also possible to override all that by declaring an explicit rectangle (or the client rectangle) in which you want the image to be drawn. You saw how you can scale an image, change its size, or change its resolution. All of these issues affect the quality of the displayed image.

This chapter presented just a few of the many different ways you might want to manipulate an image: by cropping, skewing, reflecting, rotating, or cloning it. You also saw how to get thumbnail information from an image.

Next, you saw how a technique called double-buffering can be used to improve performance, particularly in a situation where you're drawing complex images at runtime. Double-buffering takes advantage of the layering effect to reduce the amount of work that needs to be done, and it makes the whole drawing task more manageable (by breaking the image down into related groups of elements), and thus also improves the rendering (particularly on lower-powered machines). We also revisited the subject of alpha here, particularly as it relates to layering one image over another.

You learned about a few other useful techniques, too. You used the `Graphics` object's `Draw` methods to control the color and alpha of individual pixels in the image, and saw a handy technique for working with animated images (such as animated GIFs). You used the `Bitmap.Save` method to save a file to a specified image format (and saw, in the process, how it allows you to convert an image from one format

to another). You also saw how you can check for support of certain image formats by using the `ImageCodecInfo` class to confirm the existence of the appropriate codec on the system.

We've covered most of the image-related functionality you should need when you're creating custom controls. In the [next chapter](#), we'll turn our attention to the subject of graphics paths and regions.

 Previous

Next 

 PreviousNext 

Chapter 6: GraphicsPaths and Regions

Overview

In the earlier chapters of this book, you learned how to use pens and brushes (in conjunction with the `Draw...` and `Fill...` methods of the `Graphics` class) to create abstract regular shapes on the page. In the examples, we drew and filled lines, rectangles, regular curves, and ellipses. In this chapter, we'll take a look at two classes that take this type of drawing a significant step further: `GraphicsPath` and `Region`. These classes enable you to work with more irregular shapes and to group sets of shapes together for more effective processing.

For example, the `GraphicsPath` class would be useful for creating a computer-aided design (CAD) application. If you are writing an application to create electrical blueprints of a building, you may have specific types of items that you want to place repeatedly on your drawing, such as light switches, fixtures, switchgear, or core drills. A small drawing would represent each of these. By encapsulating each small drawing in a `GraphicsPath` object, you could subsequently draw each one at any location with one call into GDI+.

Recently, I needed to implement a custom control that had a practical use of the `Region` class. In this custom control, I needed to allow users to edit text. For this application, I found it necessary to write my own edit control for editing the text. This edit control behaved in the same way as other Windows edit controls, allowing the users to select text and delete, cut, paste, or replace it. The best way to implement selected text is to set up a clipping region, exclude the rectangle that contains the selected text, and draw the normal text (by default, as black on white). Then set up another clipping region, intersect the rectangle that contains the selected text, and draw the text as white on black. The use of these more elaborate clipping regions makes it easy to draw the selected text in the edit control.

This chapter covers the basic features of `GraphicsPath` and `Region` objects. Specifically, we'll focus on how to use these classes to compose outlines and shapes, and to perform some simple operations with them. In the [next chapter](#), you'll see how these classes can be used in operations such as clipping, invalidation, and creating custom gradient brushes.

 PreviousNext  PreviousNext 

An Overview of the `GraphicsPath` and `Region` Classes

`GraphicsPath` and `Region` are two classes that contribute significantly to the power of GDI+ as a graphics-programming interface. Together, they allow you to build higher levels of abstraction, and thereby create simpler, cleaner code. These two classes are intimately related, and yet quite distinct in their purposes and capabilities.

A `GraphicsPath` class represents a set of subpaths, or *figures*. Each figure is a connected path, composed of a series of line and curve segments and geometric figures. These segments and geometric figures can be described in any of the standard ways you've seen in this book so far: such as arcs, Bézier curves, ellipses, rectangles, and so on. To create a figure, you simply list the components of the figure in the right order. The `GraphicsPath` class computes the figure to be the outline that results when these component outlines are joined together in the order specified. (You'll see some examples of this in the [next section](#).)

The `GraphicsPath` itself is the path composed of the ordered set of figures. Because the set of figures is ordered, and because each figure is composed of an ordered set of lines and curves, the `GraphicsPath` is a path with a beginning and an end. The figures within a `GraphicsPath` object don't need to be connected to one another. This means that the figures don't need to be top-to-tail on the drawing surface. In fact, usually, they're *not* top-to-tail, and thus the path described by the `GraphicsPath` object is discontinuous. To follow such a path from one end to the other, you would need to "leap through space" from the end of one figure to the beginning of the next.

The `Region` class is similar to the `GraphicsPath` class, but it relates to areas, or regions, instead of paths. Thus, you can use a `Region` object to represent a shape or a set of shapes. A region can consist of geometric shapes, like rectangles and ellipses, or custom shapes whose outlines are composed of whatever lines and curves you care to specify. Like a `GraphicsPath`, the shapes that make up a `Region` don't necessarily need to be connected. (You'll see some examples in the "[Using the Region Class](#)" section later in this chapter.)

Once you've created and defined a `GraphicsPath` object, you can draw it to your drawing surface using a `Pen` object. This will draw just the outline of the `GraphicsPath` to your drawing surface. After constructing a `Region` object, you can fill the region using a `Brush` object. In both cases, you can use the `GraphicsPath` and `Region` classes to compose complex paths and shapes, but the task of rendering the completed path or shape to the screen can be achieved *in a single method call*. This makes the rendering process much tidier and also allows you to organize your code more efficiently.

Let's start to look at some real `GraphicsPath` and `Region` examples, so you can get a better idea of how they work.

 Previous

Next 

 Previous

Next 

Using the `GraphicsPath` Class

The `GraphicsPath` class encapsulates a connected series of lines and curves. You may recall that we used a `GraphicsPath` object back in [Chapter 3](#), to demonstrate the effect created by a `PathGradientBrush`.

Essentially, you use a `GraphicsPath` object when you want to describe an outline. Once you've described the outline, you can do a number of things with it: draw it using a pen, fill its interior using a brush, or use it to create clipping regions. You'll see all these applications during this chapter and the next one. But first, let's start with some simpler examples to introduce the basics.

Composing a `GraphicsPath`

Before you can do anything with a `GraphicsPath` path, you need to compose it. Once you've instantiated a new `GraphicsPath` object, you compose the `GraphicsPath` path segment by segment, and figure by figure. The order in which you specify the segments and figures defines the order and direction of the path, and hence the starting and ending points of the path. Let's look at an example that creates and draws a `GraphicsPath` object.

Note If you're trying out these examples, remember that the `GraphicsPath` class is part of the `System.Drawing.Drawing2D` namespace, so you'll need a `using` directive to include it.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a new GraphicsPath object
    GraphicsPath gp = new GraphicsPath();

    // Create a figure
    gp.AddLine(10, 10, 10, 50);
    gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
    gp.AddLine(30, 70, 60, 70);
    gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
    gp.AddLine(90, 50, 90, 30);
    gp.AddLine(90, 30, 120, 10);
    gp.AddLine(120, 10, 150, 10);
    gp.AddLine(150, 10, 170, 30);
    gp.AddLine(170, 30, 170, 70);
```

```
// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
}
```

As you can see in [Figure 6-1](#), we've created a path that contains two figures. We've composed the first figure by adding a series of line and Bézier curve segments to the `GraphicsPath` object, using the `AddLine` and `AddBezier` methods. It begins with a line segment from the point (10, 10) to (10, 50), and then continues with a Bézier curve round to (30, 70), followed by another line segment, another Bézier curve, and then five more line segments, ending on the point (170, 70).

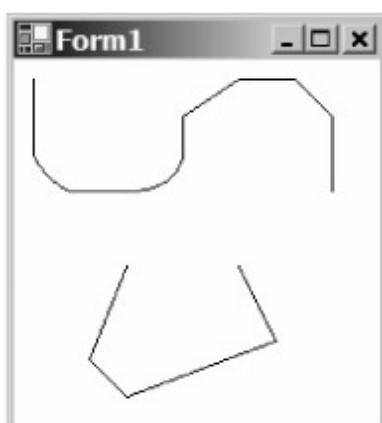


Figure 6-1: A `GraphicsPath` path

When you call the `StartFigure` method, you begin a second figure. By definition, this means that you have finished composing the first figure. As you can see, the second figure is composed of four lines, beginning at (60, 110) and ending at (120, 110).

The whole path starts at point (10, 10) and ends at point (120, 110). It's not a continuous line. In order to follow it from one end to the other, you must jump from the end of the first figure to the start of the second.

Notice how we composed the two figures in this example. To begin describing the first figure, we just launched into a sequence of `Add...` methods, which sequentially build up the first figure. We must finish describing the first figure before we begin describing the second figure. This is because as soon as you call `StartFigure` to begin describing the second figure, GDI+ assumes that you've *finished* describing the first figure. There's no going back to it! You can call `StartFigure` as often as you like. Each time you call it, you end the existing figure and begin a new one.

The `GraphicsPath` class itself doesn't provide any functionality for *rendering* the path you've created. If you want to render the path contained in a `GraphicsPath` object, you need to call a method in the `Graphics` class and pass the `GraphicsPath` object as an argument. (As you'll see later, the `Region` object is very similar in this respect.)

Open Figures and Closed Figures

A figure is *closed* if its starting point and ending point are the same. If a figure is not closed, then we say that it's *open*. (It's just like a corral—if you don't close the corral, the sheep can get out.) In the previous example, the first figure starts at (10, 10) and ends at (170, 70), and so it's open. The second figure starts at (60, 110) and ends at (120, 110), and so it is also open.

You can close a figure in a number of ways. The obvious way is explicitly to add a line or curve segment

at the end of the figure, whose endpoint is the same as the starting point of the figure. An easier way is to use the `CloseFigure` method when you've finished describing the figure, like this:

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);

// Close this figure
gp.CloseFigure();

// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

[Figure 6-2](#) shows the result. Now, the first figure is composed of a line from (10, 10) to (10, 50), a Bézier curve to (30, 70), a line to (60, 70), and a line that *closes* the figure by joining this last point back to the first. In fact, the `CloseFigure` method not only closes the existing figure, but it also starts the next figure automatically, so that the call to `StartFigure` here (for the second figure) is actually superfluous.

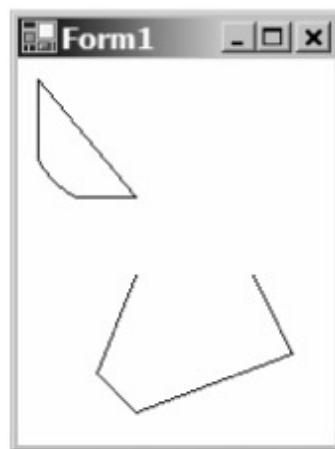


Figure 6-2: A closed figure

Alternatively, you can call the `CloseAllFigures` method, which closes all the figures in the path, up to and including the current one.

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);

// Create another figure
gp.StartFigure();

gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
```

```
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Close all figures
gp.CloseAllFigures();

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

This code creates two open figures, and then closes them both at the same time. Finally, it draws the two closed figures to the drawing surface, as shown in [Figure 6-3](#).

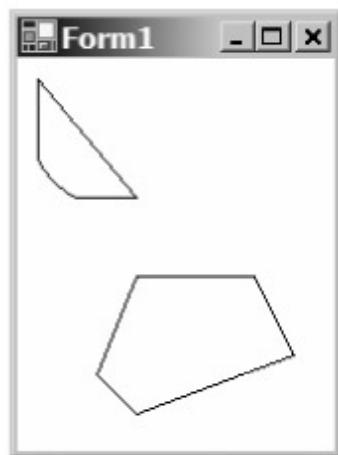


Figure 6-3: Two closed figures

You can also add geometric figures to the path. By nature, these figures are already closed, so the `CloseFigure` and `CloseAllFigures` methods have no effect on these figures. Here is an example:

```
// Create a new GraphicsPath object
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddRectangle(new Rectangle(10, 50, 80, 20));

// Create another figure
gp.AddEllipse(50, 10, 20, 80);

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();
```

In this example, the first figure is a rectangle and the second is an ellipse, as shown in [Figure 6-4](#). Note that you don't need to use `StartFigure` to end the first figure and start the second, because the first figure is already closed (and therefore complete).

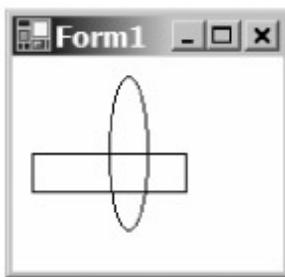


Figure 6-4: Path of geometric figures

In short, if the first and last points of a figure are different, and you don't call the `CloseFigure` or the `CloseAllFigures` method, then the figure will be open.

Absolute Coordinates and a Shorthand for Straight-Line Figures

Note that the arguments to the `GraphicsPath.AddLine` method (and other `Add...` methods) are not relative. They represent the *absolute* coordinates of the starting point and ending point. Thus, the operation `gp.AddLine(50, 0, 0, 20)` will add a line that runs between the points (50, 0) and (0, 20), regardless of what you've added to the figure so far.

This raises the interesting question of what happens when consecutive `Add...` method calls describe line or curve fragments that don't meet one another. For instance, in the first example, we created a nine-segment figure like this:

```
// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);
gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
gp.AddLine(90, 50, 90, 30);
gp.AddLine(90, 30, 120, 10);
gp.AddLine(120, 10, 150, 10);
gp.AddLine(150, 10, 170, 30);
gp.AddLine(170, 30, 170, 70);
```

Here, we've carefully specified the arguments of these `Add...` methods, so that each one runs on from the previous one—segments n and $n+1$ meet at (10, 50), (30, 70), (60, 70), (90, 50), (90, 30), (120, 10), and so on. What happens if you're not so careful?

In fact, if the end of one segment is not the same as the beginning of the next segment, GDI+ assumes that you want to join them with a straight-line segment. What this means is that, if you wish, you can specify figures with straight-line segments by omitting some of the `AddLine` operations (just as long as you don't omit any of the coordinate information). For example, you can describe the sample nine-segment figure with three fewer `AddLine` operations, like this:

```
// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
// gp.AddLine(30, 70, 60, 70);
gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
gp.AddLine(90, 50, 90, 30);
// gp.AddLine(90, 30, 120, 10);
gp.AddLine(120, 10, 150, 10);
// gp.AddLine(150, 10, 170, 30);
gp.AddLine(170, 30, 170, 70);
```

Filling the Area Bounded by a `GraphicsPath`

As you'll see a little later in this chapter, you can use a `GraphicsPath` as the boundary of a `Region`, but only if all the figures in the `GraphicsPath` are closed. Moreover, you can apply a `Brush` object to fill the area bounded by your `GraphicsPath` object, like this:

```
GraphicsPath gp = new GraphicsPath();
```

```

// Create an open figure
gp.AddLine(10, 10, 10, 50);
gp.AddLine(10, 50, 50, 50);
gp.AddLine(50, 50, 50, 10);

// Start a new figure
gp.StartFigure();
gp.AddLine(60, 10, 60, 50);
gp.AddLine(60, 50, 100, 50);
gp.AddLine(100, 50, 100, 10);
gp.CloseFigure();

// Add a geometric shape (a rectangle) to the path
Rectangle r = new Rectangle(110, 10, 40, 40);
gp.AddEllipse(r);

// Fill the area 'bounded' by the path
g.FillPath(Brushes.Orange, gp);

// Draw the path
g.DrawPath(Pens.Black, gp);

// Clean up
gp.Dispose();

```

Of course, it doesn't make sense to fill an open area, like the one we've created here. In this case, the `FillPath` method works out the closed path that you would get if you called `CloseAllFigures` on the `GraphicsPath`. Then it uses the specified Brush to fill the area bounded by that closed path, as shown in [Figure 6-5](#).

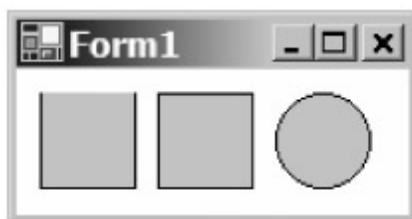


Figure 6-5: Filled paths

GDI+ doesn't actually call `CloseAllFigures` on the `GraphicsPath` itself. It first creates a copy of the `GraphicsPath`. Then it calls `CloseAllFigures` on the copy and uses *that* path to perform the filling operation.

Specifying GraphicsPath Properties and Methods

GDI+ offers a number of methods and properties that give you quite a bit of power when working with paths. [Table 6-1](#) describes the five public properties of the `GraphicsPath` class.

Table 6-1: GraphicsPath Properties

Property	Description
FillMode	If you're filling the interior of the path, this property dictates how it will be filled. The mode value (from the <code>FillMode</code> enumeration) can be <code>Alternate</code> and <code>Winding</code> . You can get or set the value of this property. The exact way that these fill modes fill the interior of a path can depend on the orientation of the path (the starting and ending points of the path, and the resulting direction of the path). So, the fill behavior of some brushes can depend on the orientation of the path, and therefore on the order in which you describe the individual line and curve segments and the individual figures. This is relevant only when the path crosses itself. If the path doesn't cross itself, the <code>FillMode</code> property doesn't matter. When building custom controls, in most situations, you will not need to create a path that crosses itself.

PathPoints	This property gets the points in the path, in the form of an array.
PathTypes	Each point in a <code>GraphicsPath</code> has a type (for example, <code>Start</code> , <code>Line</code> , <code>Bezier</code> , or <code>CloseSubpath</code>). This property gets the types of the points in the path, in the form of an array. Note that all the possible point types are listed in the <code>PathPointType</code> enumeration, which is part of the <code>System.Drawing.Drawing2D</code> namespace.
PointCount	This property gets the number of elements in the arrays returned by the <code>PathPoints</code> or <code>PathTypes</code> properties.
PathData	This property gets a <code>PathData</code> object that contains information about how the path is constructed. The <code>PathData</code> object has two properties— <code>Points</code> and <code>Types</code> —which give the same arrays as those described for the <code>PathPoints</code> and <code>PathTypes</code> properties.

Most of the methods in the `GraphicsPath` class facilitate construction of your path. These methods, which are all prefixed with the word `Add`, allow you to add things like lines, curves, and geometrical figures to the path. You've seen these at work in the previous examples. Here's the full list of the `Add`. . . methods:

AddArc	AddClosedCurve	AddLine	AddPie	AddRectangles
AddBezier	AddCurve	AddLines	AddPolygon	AddString
AddBeziers	AddEllipse	AddPath	AddRectangle	

Table 6-2 lists some of the other methods of the `GraphicsPath` class. This table is not intended to be a complete reference, but rather is a guide to the different ways that you can construct a path.

Table 6-2: Some `GraphicsPath` Methods

Method	Description
Clone	Creates a <code>GraphicsPath</code> object that contains a copy of the current path.
CloseFigure	Closes the current figure by adding a line between the ending point of the last segment and the starting point of the first segment. Automatically starts a new figure in the path.
CloseAllFigures	Closes all figures drawn so far, by adding lines between each figure's ending point and its start point. Automatically starts a new figure in the path.
Flatten	Converts curves into a series of connected line segments.
GetBounds	Returns a rectangle that encloses the path.
IsOutlineVisible	Determines whether a particular point is contained in the outline of the path. Could be used for hit testing.
IsVisible	Determines whether a particular point is contained in the interior of the path.
Reset	Clears the path.
Reverse	Reverses the order of the points in the path.
SetMarkers and ClearMarkers	Set and clear <i>markers</i> , which are used to separate groups of subpaths. You can use <code>SetMarkers</code> to insert a marker at a specific point in a path as you are constructing a path. Later, you can create a new path that consists of a <i>subset</i> of the original path that lies between two markers in that path. You can use <code>ClearMarkers</code> to clear any markers from an existing path.
StartFigure	Starts a new figure in the path. Leaves the previous figure open.
Transform	Applies a transform matrix to the path.
Warp	Applies a warp transform to the path. A warp transform is defined by a rectangle and a parallelogram.

Also note that the `GraphicsPath` constructor is overloaded to accept an array of points, an array of point types, or a fill mode. If your `GraphicsPath` isn't very complicated, this approach works well. See the documentation at <http://msdn.microsoft.com/library> for more information.

 Previous

Next 

 Previous

Next 

Using the Region Class

You can use a `Region` object to describe a region—one or more areas of real estate on your surface. Actually, a `Region` object doesn't even need to include one area; it may not include any areas at all. Having no areas would be analogous to the empty set (of set theory).

When you work with a `Region` object, you generally begin by defining what area the `Region` covers, and then performing operations that use the defined `Region`. In this regard, you use a `Region` object in much the same way as you use a `GraphicsPath` object. Once you've defined your `Region`, you can perform a number of operations: fill it with a `Brush` object, use it for hit testing, or use it as a clipping region (as described in the [next chapter](#)).

It makes sense to begin by seeing how you might define a `Region`. Unlike the `GraphicsPath` class, the `Region` class doesn't have an army of `Add...` methods. In fact, while a `GraphicsPath` object has its figures, there is no equivalent concept in a `Region` object. Although a `Region` object typically does consist of a number of disconnected enclosed areas, they're created in a very different way. You usually create a `Region` by using one of the overloaded `Region` class constructors. We'll look at some ways to create `Region` objects here.

Creating a Region from a GraphicsPath

One way to create a `Region` is to use a `GraphicsPath` to define a path, and then use this path as the outline of your new `Region` object. In the following example, we use a familiar `GraphicsPath` object (one from earlier in the chapter) to create a region, and then we fill the resulting region with a `Brush`.

```
// Create a GraphicsPath
GraphicsPath gp = new GraphicsPath();

// Create an open figure
gp.AddLine(10, 10, 10, 50);
gp.AddLine(10, 50, 50, 50);
gp.AddLine(50, 50, 50, 10);

// Start a new figure
gp.StartFigure();
gp.AddLine(60, 10, 60, 50);
gp.AddLine(60, 50, 100, 50);
gp.AddLine(100, 50, 100, 10);
gp.CloseFigure();

// Add a geometric shape (a rectangle) to the path
Rectangle r = new Rectangle(110, 10, 40, 40);
gp.AddEllipse(r);

// Create a Region whose boundary is the above GraphicsPath
Region reg = new Region(gp);

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
gp.Dispose();
```

[Figure 6-6](#) shows the result. The shapes that you see are familiar, but the way you've filled them is

different. The `Region` class constructor we used is one that accepts a `GraphicsPath` object. It takes a copy of the path, closes it, and then uses it to create a `Region`. As you can see, this region is composed of three disconnected areas: two four-sided polygons and a circular ellipse.

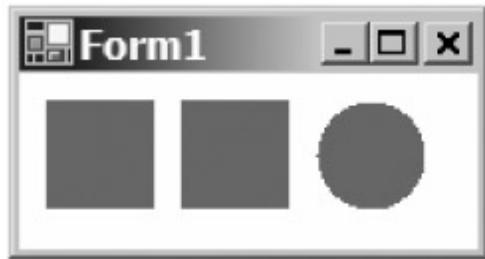


Figure 6-6: Filled Region

So, `Region` objects (like `GraphicsPath` objects) don't need to be connected or contiguous. Moreover, `GraphicsPath` objects give you a way to describe areas whose boundaries are irregular.

Here's another example using a `GraphicsPath`:

```
// Create a GraphicsPath
GraphicsPath gp = new GraphicsPath();

// Create a figure
gp.AddLine(10, 10, 10, 50);
gp.AddBezier(10, 50, 10, 55, 25, 70, 30, 70);
gp.AddLine(30, 70, 60, 70);
gp.AddBezier(60, 70, 85, 70, 90, 55, 90, 50);
gp.AddLine(90, 50, 90, 30);
gp.AddLine(90, 30, 120, 10);
gp.AddLine(120, 10, 150, 10);
gp.AddLine(150, 10, 170, 30);
gp.AddLine(170, 30, 170, 70);

// Create another figure
gp.StartFigure();
gp.AddLine(60, 110, 40, 160);
gp.AddLine(40, 160, 60, 180);
gp.AddLine(60, 180, 140, 150);
gp.AddLine(140, 150, 120, 110);

// Create a Region whose boundary is the above GraphicsPath
Region reg = new Region(gp);

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
gp.Dispose();
```

As you can see in [Figure 6-7](#), this `Region`'s outline is a familiar shape—its outline is the `GraphicsPath` we drew at the beginning of this chapter. This time, the top figure is not entirely concave, so when GDI+ closes the figure to create the `Region`, the resulting interior is in two pieces.

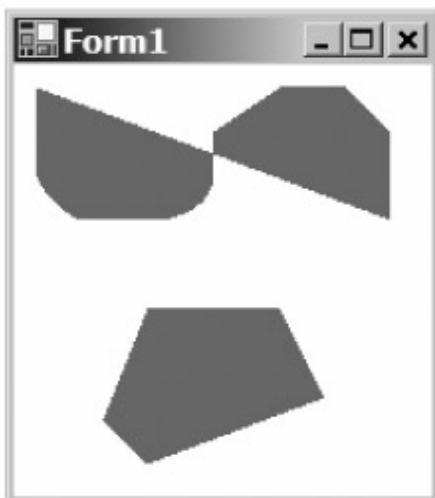


Figure 6-7: Another filled Region

Creating a Region from a Rectangle

You can create a Region from a Rectangle object, similar to how you create one from a GraphicsPath. Here's an example:

```
// Create a Region whose boundary is the above Rectangle
Region reg = new Region(new Rectangle(10, 10, 80, 20));

// Fill the Region
g.FillRegion(Brushes.Green, reg);

// Clean up
reg.Dispose();
```

As shown in [Figure 6-8](#), the result isn't very surprising. However, it's handy to have an easy way to put together a rectangular Region, because many of the Region objects you're likely to create will be rectangular.



Figure 6-8: Rectangular Region

Creating a Region from Another Region

To create a Region from an existing Region, the constructor doesn't accept the existing Region object itself; instead, it expects an array of data about the Region. You can get the required data about such a Region from its GetRegionData method, which returns the required data in the form of a RegionData object. You can then pass this object into the Region constructor, and hence create a new Region object just like the existing one, as follows:

```
Region r1 = new Region(new Rectangle(10, 10, 80, 20));
RegionData r1Data = r1.GetRegionData();
Region r2 = new Region(r1Data);
```

Alternatively, it's easier just to use the Clone method instead, like this:

```
Region r1 = new Region(new Rectangle(10, 10, 80, 20));
Region r2 = r1.Clone();
```

Performing Union and Intersection Operations on Regions

Once you've constructed a region, you can modify it. The `Region` object provides five methods that allow you to perform *set algebra* on the region. This involves specifying a second area (in the form of a `Rectangle`, a `RectangleF`, another `Region`, or the internal of a `GraphicsPath`), and then performing an operation. This results in a new `Region` that is a subset of the two.

Here's an example:

```
// Create two rectangles
Rectangle rect1 = new Rectangle(50, 10, 50, 130);
Rectangle rect2 = new Rectangle(10, 50, 130, 50);

// Set Region using first rectangle
Region reg = new Region(rect1);

// Modify Region to be intersection of itself with second rectangle
reg.Intersect(rect2);

// Draw the result so we can see it
g.FillRegion(Brushes.Orange, reg);
g.DrawRectangle(Pens.Black, rect1);
g.DrawRectangle(Pens.Black, rect2);
```

In this example, we first set the `Region` object to represent the area inside the tall rectangle. Then we modify it—we reduce its size by taking only the intersection of the `Region` with the `Region` described by `rect2`. The resulting `Region` is shown in the shaded area in [Figure 6-9](#). The figure also shows the two rectangles.

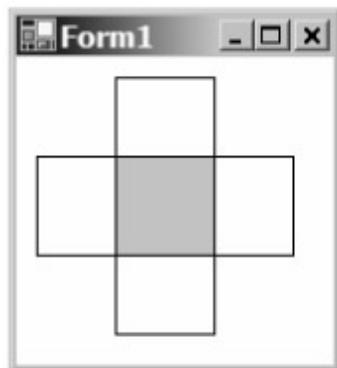


Figure 6-9: Intersected Regions

The five set methods, which are listed in [Table 6-3](#), represent five different set algebra operations. They all work in a similar way.

Table 6-3: Region Set Algebra Operations

Operation	Description
<code>r1.Intersect(r2);</code>	Updates <code>r1</code> to be the intersection between <code>r1</code> and <code>r2</code> (that is, any portion that is in both <code>r1</code> and <code>r2</code>)
<code>r1.Union(r2);</code>	Updates <code>r1</code> to be the union of <code>r1</code> and <code>r2</code> (that is, any portion that is in either <code>r1</code> or <code>r2</code> or both)
<code>r1.Xor(r2);</code>	Updates <code>r1</code> to be the exclusive union of <code>r1</code> and <code>r2</code> (that is, any portion that is in either <code>r1</code> or <code>r2</code> but not both)
<code>r1.Complement(r2);</code>	Updates <code>r1</code> so that it <i>includes</i> areas that are contained within <code>r2</code> , but <i>excludes</i> any portion that was originally contained in <code>r1</code>
<code>r1.Exclude(r2);</code>	Updates <code>r1</code> to <i>exclude</i> any portion that is also contained within <code>r2</code>

Figure 6-10 demonstrates all five set operations. In each of these, *r1* is the tall, thin rectangle and *r2* is the short, wide one.

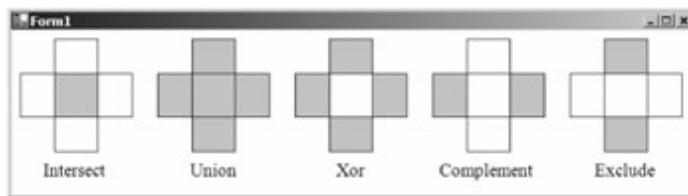


Figure 6-10: Results of set operations

Note The code to create [Figure 6-10](#) is included with the downloadable code for the book, available from the Downloads section of the Apress web site (www.apress.com). It is quite similar to the code used to create [Figure 6-9](#).

Using Other Region Methods

You can use a number of other methods to manipulate and query Region objects (and return information about them). Some of them are listed in [Table 6-4](#). For a full list, see the documentation at <http://msdn.microsoft.com/library>.

Table 6-4: Some Region Methods

Method	Description
Clone	Returns a Region object that is a copy of the existing one.
GetBounds	Gets a Rectangle object that bounds this Region object on the specified drawing surface.
GetRegionData	Gets a RegionData object that contains data that defines the region. You can use the RegionData object to create a new Region.
GetRegionScans	Gets an array of RectangleF objects that approximate this Region.
IsEmpty	Returns a Boolean: true if the Region is empty on the specified drawing surface; false otherwise.
IsInfinite	Returns a Boolean: true if the Region has an infinite interior; false otherwise.
IsVisible	Returns a Boolean: true if any part of the Region is visible when drawn on the drawing surface of the specified Graphics object; false otherwise. A number of overloads for this method test for visibility of an intersection of the Region with a given point or rectangle on the given drawing surface.
MakeEmpty	Initializes this Region so that its interior is empty.
MakeInfinite	Initializes this Region so that its interior is infinite.
Transform	Applies a transform matrix to the Region.
Translate	Offsets the Region by an X and Y offset.

Here, we'll look at some examples of using these methods to get information and to work with infinite regions. You'll see some of the other methods in use in later chapters.

Region Information

The following example demonstrates the methods that return Booleans:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
```

```
// Create a Region object, cut a rectangular hole in it, and fill it
Region r = new Region(new Rectangle(30, 30, 30, 60));
r.Exclude(new Rectangle(40, 40, 10, 10));
g.FillRegion(Brushes.Orange, r);

// Tell us about the Region
Console.WriteLine("This Region: ");
Console.WriteLine(r.IsInfinite(g) ? " - is infinite"
                           : " - is finite");
Console.WriteLine(r.IsEmpty(g) ? " - is empty"
                           : " - is non-empty");

PointF pf = new PointF(35.0f, 30.0f);
Console.WriteLine((r.IsVisible(pf) ? " - includes"
                           : " - excludes")
                  + " the point (35.0, 50.0) ");

Rectangle rect = new Rectangle(25, 65, 15, 15);
g.DrawRectangle(Pens.Black, rect);
Console.WriteLine((r.IsVisible(rect) ? " - is visible"
                           : " - is invisible")
                  + " in the rectangle shown" );

r.Dispose();
}
```

When you run this code, you get a list of output to the console window:

```
This Region:
- is finite
- is non-empty
- includes the point (35.0, 50.0)
- is visible in the rectangle shown
```

You also see the Region in the client window, as shown in [Figure 6-11](#).

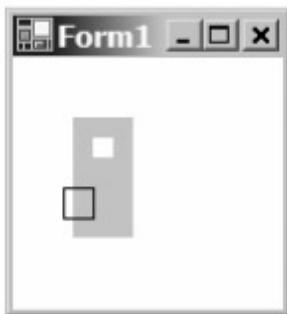


Figure 6-11: Filled Region with a rectangular hole

As you can see, the filled Region is finite, in that it doesn't extend in all directions as you see it in the drawing surface. It's also nonempty—it contains at least one point. We've selected the point (35, 50) to see whether it's contained within the Region, and the output tells us that it is. Finally, the output confirms that the Region is visible through the rectangle shown, because the interior of the rectangle contains a part of the Region.

Infinite Regions

It's sometimes useful to set a Region to be equal to the entire drawing surface, extending infinitely in both directions. Even if you enlarge the drawing surface, the Region continues to include all of it. You then can use the set algebra operations (`Intersect`, `Union`, `Xor`, `Complement`, and `Exclude`) to cut out bits of the infinite Region and get your desired Region.

For example, you might use this technique when you want to draw a background over the entire custom control, apart from certain rectangles or areas in which you want to draw other elements. In this case,

making an infinite Region by using the `MakeInfinite` method, and then cutting rectangles out of it is a very convenient approach. Here is an example:

```
// Create a Region object and set it to be infinite
Region r = new Region();
r.MakeInfinite();

// Now cut a rectangular hole in it and draw the result
r.Exclude(new Rectangle(30, 30, 50, 20));
r.Exclude(new Rectangle(30, 60, 50, 20));
g.FillRegion(Brushes.Orange, r);

r.Dispose();
```

Here, we've set the `Region` object to be the infinite space, before cutting out a couple of rectangles and filling what's left. [Figure 6-12](#) shows the result.



Figure 6-12: Region with pieces cut out

[Previous](#)

[Next](#)

[Previous](#)

[Next](#)

Summary

This chapter covered the `GraphicsPath` and `Region` classes. A `GraphicsPath` object is used to represent a path. This path is composed of an ordered set of figures, and each figure is composed of an ordered set of line segments, curve segments, and geometric figures. To define the `GraphicsPath` object, you build up the path piece by piece. A `Region` object, by contrast, represents an area. You cannot define a `Region` piece by piece. More commonly, you describe a `Region` by using set algebra operations (`Intersect`, `Union`, `Xor`, `Complement`, and `Exclude`). `GraphicsPath` and `Region` objects can both cope with being disconnected or discontinuous, and a `Region` can be infinite.

You saw how each of the figures in a `GraphicsPath` is either open or closed, and that you can call the `CloseFigure` method or `CloseAllFigures` method to close open figures. You can use a `GraphicsPath` as the outline of a `Region`. If the `GraphicsPath` contains any open figures, GDI+ works out what the closed path would look like, and then builds the `Region` from that.

You can draw a `GraphicsPath`'s path using a `Pen` object, and you can fill a `Region` using a `Brush` object. (As a sort of shorthand, you can also fill the `Region` whose boundary is a given `GraphicsPath` directly, by passing the `GraphicsPath` and a `Brush` to the `Graphics.FillPath` method.)

Both the `GraphicsPath` and `Region` classes have many utility methods and properties that give you a great deal of power and flexibility.

In the [next chapter](#), you'll see how the `GraphicsPath` and `Region` classes can be very useful in the context of clipping. We'll also take a look at invalidation.

[Previous](#)

[Next](#)

 PreviousNext 

Chapter 7: Clipping and Invalidation

Overview

In the [previous chapter](#), we looked at how to use a `Region` object to describe a portion of a graphic. This portion or region may consist of a single "fenced-off" area or a union of several disconnected areas. You saw that you can define a region using a `Region` object directly, or you can define it indirectly by using a `GraphicsPath` object to describe the outline of the region.

Regions are quite versatile, and you can apply them in a wide variety of ways. For example, you can use regions for clipping, which allows you to restrict the area of a drawing surface that your drawing operations can be applied to (and hence force the rest of the surface to remain unaffected). This technique allows you to draw one region of the graphic at a time, and to be very selective about what goes into that region.

You can also use a region to control (and potentially to reduce) the amount of work that the operating system needs to do when it renders your graphic to the screen. If the size of the graphic is considerably larger than the client window, you can use the relative sizes of each so that the operating system draws only the visible parts of your graphic. Moreover, when the window size changes and the graphic needs to be redrawn, the *amount* of redrawing can be limited.

Regions are also helpful when a custom control that has been rendered to the screen is subsequently partially or totally obscured by another window. If that second window is then moved, your control will need to be partially or totally redrawn on the screen. The operating system can limit the amount of rerendering work it must do by calculating what region of the control has been "invalidated" by the second window, because that portion is the only part that needs to be redrawn.

Another explicit form of invalidation can also come into play when you write a control whose appearance depends on the state of the control (that is, the data values associated with the control). Sometimes, the control also allows users to perform actions that change the control's state. Whenever a user performs such an action, the result is that the display no longer reflects the actual state of the control, so it must be updated. In this case, you can write code to identify the region that has been invalidated by the user's action, and hence force a `Paint` event that repaints the invalidated region.

In this chapter, we'll look at these particular applications of regions, focusing on using clipping regions. We'll also explore the concept of invalidation. This chapter covers the following topics:

- Specifying the size and shape of the clipping region, to describe which part of a drawing surface you want to modify
- Detecting the clipping region specified on a `Paint` event, and hence minimizing the amount of rendering work to the size of the client rectangle (or smaller)
- Understanding what happens when the form needs to be completely or partially redrawn, and the operating system raises a `Paint` event
- Using invalidation to force a `Paint` event from within the application

We'll start with a discussion of the clipping region, the central tool in this chapter, and how it is implemented in GDI+.

 PreviousNext  PreviousNext 

Uses of Clipping Regions

We can define the *clipping region* to be the region of the drawing surface in which the effects of drawing actions will actually be seen. But to really appreciate this definition, consider the two different sets of circumstances in which the clipping region is used:

- Sometimes, you would like to control the clipping region explicitly in your code, and hence use it to specify and limit the region of the drawing surface to which you're drawing.
- Other times, the .NET Framework needs to control the clipping region, when it needs to repaint portions of a form or the entire form.

Let's look at our definition in the light of these two different contexts, to understand what's really going on. First, you'll see how to crop with clipping regions.

 Previous

Next 

 Previous

Next 

Cropping with the Clipping Region

In the GDI+ object model, the *clipping region* is a property of the `Graphics` object. This is because when you specify the clipping region in your code, it relates to the way that your drawing actions are represented on the drawing surface itself. You can use this to control the clipping region programmatically, and hence control *precisely* which region of the drawing surface is affected by a drawing operation.

It's useful to compare this to a piece of paper and a stencil. Think of the drawing surface as a piece of paper and the clipping region as the stencil. Lay the paper on a table, and then lay the stencil over the paper and fix it in place. Once the stencil is in place, you can draw only onto certain regions of the paper: the regions that are exposed through the cutouts of the stencil. Similarly, in GDI+, when you apply a clipping region to a drawing surface, subsequent drawing operations apply only to the regions of the drawing surface exposed by the clipping region.

Let's begin with a simple example that shows this stencil effect.

Setting the Clipping Region

The GDI+ `Graphics` object provides a number of methods and properties that you can use to manipulate the clipping region. In the names of these methods and properties, they simply refer to the clipping region as the *clip*.

We'll use one of these methods, `SetClip`, to set the clipping region to be a rectangular region. For convenience, we'll draw the rectangle first, just so that we can see it. Then we'll apply the rectangular clipping region, and then perform some other drawing operations to see how the clipping region affects the results.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Define the rectangle that we'll use to define our clipping region
    Rectangle rect = new Rectangle(10, 10, 80, 50);

    // Draw the clipping region
    g.DrawRectangle(Pens.Black, rect);

    // Now apply the clipping region
    g.SetClip(rect);

    // Now draw something onto the drawing surface
    // (with the clipping region applied)
    g.FillEllipse(Brushes.Blue, 20, 20, 100, 100);
}
```

There are a number of important points to consider here, so let's step through the example. At the time the `Paint` event is fired, the clipping region is automatically set to be the shape and size of the client rectangle (we'll return to this in the "[Watching the ClipRectangle](#)" section later in this chapter, when we

discuss the `ClipRectangle` property).

Therefore, when we call the `DrawRectangle` method, it's like drawing onto a piece of paper without a stencil. The entire drawing surface is exposed, and so the black rectangular outline is drawn to the drawing surface as shown in [Figure 7-1](#).

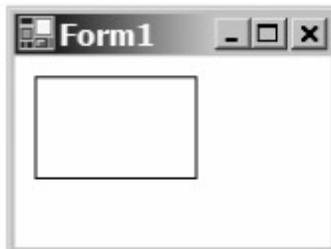


Figure 7-1: Clipping region outline

Next, we use the `SetClip` method to set the clipping region to be the region defined by our `Rectangle` object. This is like applying a rectangular stencil to the drawing area. Subsequent drawing operations will affect *only* the region of the drawing surface that is exposed by the clipping region stencil. Thus, when we draw the blue, circular ellipse, the drawing surface gets to see *only* the part of the ellipse that is contained within the clipping region, as shown in [Figure 7-2](#).

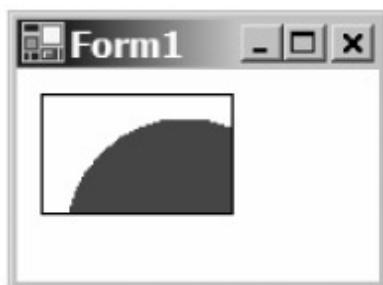


Figure 7-2: Clipped ellipse

In this context, we're effectively using the clipping region to *crop* a sequence of drawing operations, so that they have no effect outside the clipping region. This is often the only way to draw a partial graphic.

Irregular-Shaped Clipping Regions

In the previous example, the clipping region we chose was rectangular. In fact, when you're using the clipping region via the `Graphics` object like this, you can set it to be any `Region` object you like.

For example, let's create a very irregularly shaped region, use it to set the clipping region, and then perform a drawing operation.

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create a Region object and make sure it's empty
    Region reg = new Region();
    reg.MakeEmpty();

    // Create an irregular region
    // ... add a circle to the region
    GraphicsPath gp = new GraphicsPath();
    gp.AddEllipse(10, 10, 50, 50);
    reg.Union(gp);
    // ... add a triangle to the region
    gp.Reset();
    gp.AddLine(40, 40, 70, 10);
    gp.AddLine(70, 10, 100, 40);
```

```
gp.CloseFigure();
reg.Union(gp);
// ... add a rectangle to the region
reg.Union(new Rectangle(40, 50, 60, 60));

// Set the clipping region
g.SetClip(reg, CombineMode.Replace);

// Paint the client rectangle green.
// Only the clipped region will be affected
g.FillRectangle(Brushes.Green, this.ClientRectangle);

gp.Dispose();
reg.Dispose();
}
```

In this example, we create a rather complex `Region` object by using the `Union` method, and then we assign this strangely shaped region to the clipping region (using `SetClip`). Here, `union` is used as it is in set theory. (If you time-warp back to math class when you were about 12 years old, you will recall terms such as *union* and *intersection* as applied to sets.)

Note We call the `Region` object's `MakeEmpty` method at the beginning of the example because the overloaded `Region` constructor we've used (with no arguments) generates a `Region` object that represents the entire, infinite drawing surface. However, what we want to do here is start with an empty `Region` and add to it. Therefore, to create the empty `Region`, we use `MakeEmpty`.

Finally, we paint the entire client rectangle green. However, only the region exposed by the clipping region is painted green. Once again, the clipping region acts as a stencil. The result is shown in [Figure 7-3](#).

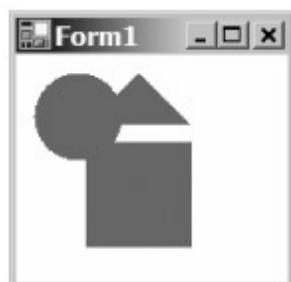


Figure 7-3: A more complex clipping region

Text for the Clipping Region

You can create a special effect by creating a `GraphicsPath` object and using its `AddString` method so that it takes on the shape of the outline of the glyphs used in the string. Then you can use the `GraphicsPath` as the outline for the clipping region, and finally paint the clipped drawing surface with whatever brush or image you choose. Here's an example:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Prepare text string
    GraphicsPath gp = new GraphicsPath();
    gp.AddString("Swirly", new FontFamily("Times New Roman"),
        (int)(FontStyle.Bold | FontStyle.Italic),
        144, new Point(5, 5), StringFormat.GenericTypographic);

    // Set clipping region
    g.SetClip(gp);
```

```
// Paint the client rectangle using an image.  
// Only the clipped region will be affected  
g.DrawImage(new Bitmap("swirly.jpg"), this.ClientRectangle);  
  
gp.Dispose();  
}
```

[Figure 7-4](#) shows the result. It looks more effective in color than here in grayscale, as you'll see if you try it out.



Figure 7-4: Text as a clipping region

SetClip Method Overloads

In the three examples so far, we've used three different overloads of the `SetClip` method. In the first example, we did this:

```
Rectangle rect = new Rectangle(10, 10, 80, 50);  
g.SetClip(rect);
```

In this case, we pass a `Rectangle` object to the `SetClip` method. Behind the scenes, the method creates a rectangular `Region` object from it and assigns it to the clipping region.

In another example, we created and built up a `GraphicsPath` object. Then we passed the `GraphicsPath` outline to the `SetClip` method, which (behind the scenes) used it to create a `Region` object and set the clipping region to that:

```
GraphicsPath gp = new GraphicsPath();  
...build the GraphicsPath...  
g.SetClip(gp);
```

In fact, there are a total of nine overloads of the `SetClip` method. You can pass in a `Rectangle`, `RectangleF`, `Graphics`, `GraphicsPath`, or `Region` object.

You can also pass in a second parameter: a value from the `CombineMode` enumeration. If you pass this second parameter, it is used to express how the specified region is to be combined with the existing clipping region. There are six values in the `CombineMode` enumeration:

- `CombineMode.Replace`
- `CombineMode.Complement`
- `CombineMode.Exclude`
- `CombineMode.Intersect`
- `CombineMode.Union`
- `CombineMode.Xor`

If some of these look familiar, it's because they use the same names as (and have a similar effect to) the five "set algebra" methods of the `Region` object, which are described in [Chapter 6](#).

Let's quickly reconsider the example in which we used the following overload of the `SetClip` method:

```
Region reg = new Region();  
...build up the region...  
g.SetClip(reg, CombineMode.Replace);
```

Here, the `SetClip` method replaces the existing clipping region with the region specified in the first parameter. If, for example, we wanted to use the intersection of the existing clipping region with the specified region, we would have used this:

```
g.SetClip(reg, CombineMode.Intersect);
```

Note When you pass a `Region` object to the `SetClip` method, you must also pass a value from the `CombineMode` enumeration. If you pass a `Rectangle`, `RectangleF`, `Graphics`, or `GraphicsPath` object, the `CombineMode` value is optional. For more on the `SetClip` method's overloads, see the documentation at <http://msdn.microsoft.com/library>.

Using Other Clipping Region Methods and Properties

In addition to the `SetClip` method, you can use four other methods within the `Graphics` class to configure the clipping region of a drawing surface. The clipping region methods are listed in [Table 7-1](#).

Table 7-1: Clipping Region Methods

Method	Description
<code>ExcludeClip</code>	Excludes a region from the existing clipping region.
<code>IntersectClip</code>	Intersects a region with the existing clipping region.
<code>ResetClip</code>	Sets the clipping region to be the infinite surface, effectively "clearing" the clipping region. This is a useful when you've finished with your custom clipping region, and you want to return to a situation where your drawing operations apply to the whole drawing surface.
<code>SetClip</code>	Sets the clipping region to be the region defined by a <code>Rectangle</code> , <code>RectangleF</code> , <code>Graphics</code> , <code>GraphicsPath</code> , or <code>Region</code> object.
<code>TranslateClip</code>	Offsets the clipping region by an X and a Y delta. You can use this method if you want to use clipping in conjunction with translation transformations (which we'll explore in Chapter 8).

Also, a few properties of the `Graphics` class facilitate clipping operations. These are listed in [Table 7-2](#).

Table 7-2: Graphics Class Clipping Properties

Property	Description
<code>Clip</code>	Sets or gets the clipping region. You can use only the <code>Region</code> class to set clipping using this property, whereas you can use a variety of other structures and classes with the <code>SetClip</code> method. Unless you set this explicitly, it is set to be the infinite drawing surface.
<code>ClipBounds</code>	Gets a <code>RectangleF</code> that bounds the clipping region for the drawing surface.
<code>IsClipEmpty</code>	Returns <code>true</code> if the clipping region for the drawing surface is empty; otherwise, it returns <code>false</code> .
<code>VisibleClipBounds</code>	Gets a <code>RectangleF</code> that bounds the visible clip region.
<code>IsVisibleClipEmpty</code>	Returns <code>true</code> if the intersection between the clipping region that you set and the clipping region of the window is empty; otherwise, it returns <code>false</code> .

As you can see, the concept of *visibility* is considered sufficiently important that you can test for it. A clipping region is visible if its intersection with the visible area of the window is nonempty.

So far, we've taken a look at some issues surrounding the notion that you can use the clipping region to create a cropping effect. Now, let's look at the second context, where the clipping region is used in relation to painting and repainting the window.

 PreviousNext 

Selective Repainting with the Clipping Region

In order to react to a `Paint` event, you have an event handler, and as you've seen, the event handler expects two arguments. Here is an example:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    ...
}
```

The second of these arguments is a `PaintEventArgs` object, which contains information about the circumstances in which the `Paint` event was raised. In particular, you can use the `PaintEventArgs` object to get a reference to the `Graphics` object that represents the drawing surface that corresponds to the `Paint` event:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Red, this.ClientRectangle);
    ...
}
```

This is all stuff that you've seen in earlier chapters of this book. The point here is that the `PaintEventArgs` object passed into the `Paint` event handler *also* contains information about the clipping region that needs to be drawn by the event handler.

Thus, when your application receives a `Paint` event, it is not only told *which* drawing surface to paint to, but also *which region* of the drawing surface needs repainting. If it is possible to avoid repainting the *entire* window, then that's great, since it means less work for the application to do.

Watching the ClipRectangle

Let's consider an example that demonstrates the usefulness of getting information about the clipping region. The following `Paint` event handler does two things each time it runs. First, it locates the specified clipping region of the specified drawing surface (both given by the `PaintEventArgs` argument) and uses the `DrawString` method to paint in the specified text. Second, it writes a line to the console window that indicates the dimensions of the rectangular region it has just painted.

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create the necessary objects to draw the text string
    String s = "When writing GDI+ code, there are three basic target ";
    s += <more text...>
    s += "be able to produce precise and effective graphics.";
    Font f = new Font("Times New Roman", 12);
    SizeF sf = g.MeasureString(s, f, 400);
    RectangleF rf = new RectangleF(10, 10, sf.Width, sf.Height);

    // Draw the text string
    g.DrawString(s, f, Brushes.Black, rf);
    f.Dispose();

    // Write info about the clipping rectangle to the console window
    Console.WriteLine("Clipping region painted: " + e.ClipRectangle);
}
```

Before we discuss this further, try it out. When you first run the example, you should see something like [Figure 7-5](#).

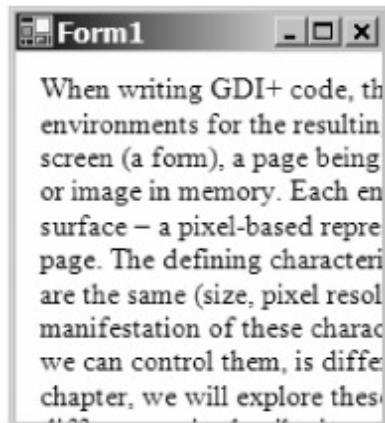


Figure 7-5: Window with text

The result on the screen is less interesting than the output to the console, which indicates the dimensions (in pixels) of the clipping region that the application needed to paint in order to produce it. You should see something like this:

```
Clipping region painted: <X=0, Y=0, Width=250, Height=200>
```

Whenever you make a change on your screen that requires this window to be repainted, it will fire a Paint event, and you'll see another line in the console window. For example, select another window and drag it so that it partially obscures the Form1 window, as shown in [Figure 7-6](#).

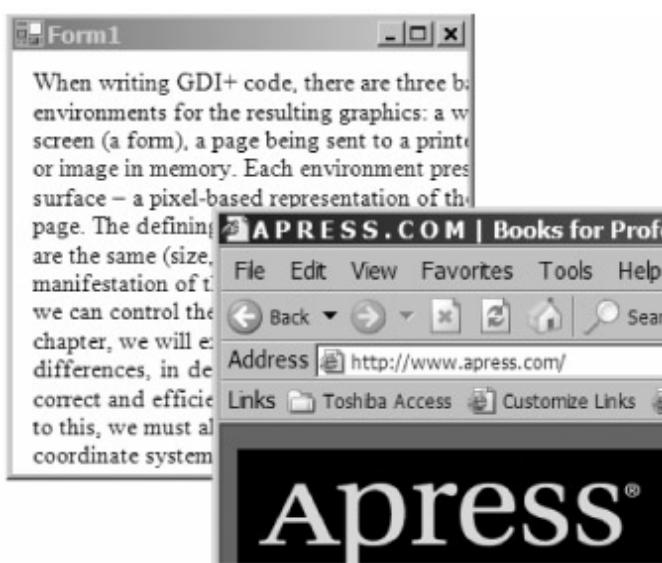


Figure 7-6: Partially obscured window

Now remove the second window so that the Form1 window is completely visible again. The operating system cannot remember what the bottom-right corner looked like, so it must ask the application by firing its Paint event. Thus, you should see another line in the console window. Here's what I got when I moved the browser window away (I'll explain the exact dimensions in a moment):

```
Clipping region painted: <X=139, Y=26, Width=111, Height=174>
```

Now, try resizing the window by dragging the bottom-right corner with your mouse. What happens when you do this depends on whether you make the window bigger or smaller.

If you make the window longer and/or wider, the operating system considers your request and works out that the window needs repainting, so it calls the Paint event of your application. We'll discuss this further in a moment. Otherwise, the operating system considers your request and works out that it has enough information about the window to redraw it *without* calling the application's Paint event. In this

case, it uses just the top-left portion of the existing client rectangle and draws that into the client rectangle of the resized window.

One of the most interesting things about this example relates to how the dimensions of the clipping rectangle are calculated. Let's look at how this is done next.

The Shape of the ClipRectangle

If you ran the preceding example and experimented with it a little, you probably noticed a couple of things. First, the clipping region passed into the `Paint` event is always a *rectangle*. But in [Chapter 6](#), I said that a `Region` object can represent almost any shape you like! Is there a contradiction here?

In fact, there is no contradiction. In general, a `Region` object doesn't have to be rectangular; it can take on any of the shapes described in [Chapter 6](#). If you use a clipping region for cropping (as discussed earlier in this chapter), you can use the `SetClip` method to set the clipping region to be any `Region` object you like.

However, when it comes to painting and repainting windows, we work in *rectangles*. This is because, generally, the operating system likes to work in rectangles, and they are easy to work with in a raster-based environment. Thus, when the `Paint` event handler is called, the clipping region is always set to be the rectangle that the operating system says it needs to repaint.

The Dimensions of the ClipRectangle

So, when the `Paint` event is fired, the clipping region will always be a rectangle. The next question is: How are the dimensions of the clipping rectangle decided?

In fact, the dimensions of the clipping rectangle are decided by the operating system that needs to render the visible part of the graphic to the drawing surface. In general, the operating system asks for the *smallest possible rectangle* that will contain all the information it needs to perform the rendering action. There are two reasons for this: it minimizes the amount of data that needs to be passed between the application and the operation system, and it also minimizes the amount of rendering work that the operating system must do to rerender the window.

So, when you first run an application, you need to render the entire window. In this case, the clipping rectangle is equal in size to the client rectangle. So, suppose your application defines the client rectangle to be 250X200 pixels:

```
private void InitializeComponent()
{
    ...
    this.ClientSize = new System.Drawing.Size(250, 200);
    ...
}
```

Then the output to the console window should confirm that the clipping rectangle is a 250X200-pixel rectangle, whose top-left corner is at (0, 0):

```
Clipping region painted: <X=0, Y=0, Width=250, Height=200>
```

If you make the window shorter and wider, then the operating system forgets about the bottom-left section of the graphic that it no longer needs. But it must fire a `Paint` event to request information about the extra rectangular region in the top-right corner, as shown in [Figure 7-7](#).

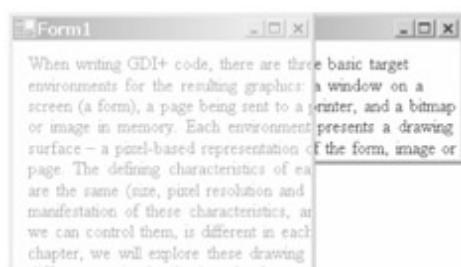


Figure 7-7: Making the window shorter and wider

In [Figure 7-7](#), the client window size was changed from 250X200 to 400X150. Therefore, the console window should confirm that the clipping rectangle is the new upper-right region:

Clipping region painted: <X=250, Y=0, Width=150, Height=150>

If you make the window taller and narrower, then the operating system forgets about the upper-right section of the graphic that it no longer needs, but fires a Paint event to request information about the extra rectangular region in the bottom-left corner. In the example shown in [Figure 7-8](#), the client window size was changed from 250X200 to 150X250, so the console window should show that the clipping rectangle is the new lower-left region:

Clipping region painted: <X=0, Y=200, Width=150, Height=50>

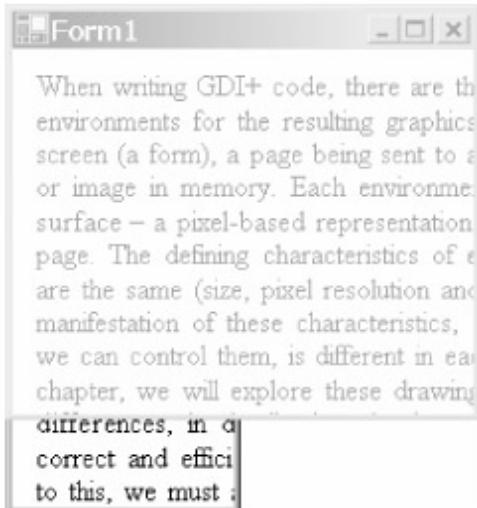


Figure 7-8: Making the window taller and narrower

If you make the window taller and wider, then it's a little different. Although the actual new area is L-shaped, the Paint event only allows the operating system to request information about a *rectangular* clipping region. The smallest possible rectangle that can cover the missing part of the graphic is the entire client rectangle. In the example shown in [Figure 7-9](#), I've changed the client window size from 250X200 to 350X300. The clipping rectangle is the same as the client rectangle:

Clipping region painted: <X=0, Y=0, Width=350, Height=300>

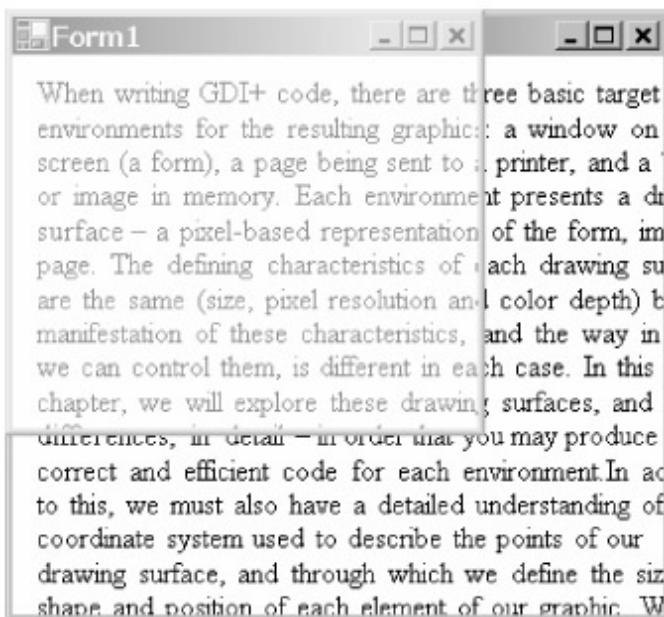


Figure 7-9: Making the window taller and wider

If you hide part of the window and then reveal it again, the operating system requests the smallest rectangle required to replace the part of the graphic that was obscured. So, in the example presented earlier ([Figure 7-6](#)), we needed a 111X174 clipping rectangle placed at (139, 26) to replace the missing

Clipping region painted: <X=129, Y=127, Width=167, Height=169>

Using the ClipRectangle to Minimize Drawing Work

The discussion in the preceding section reveals that when your form is being painted (or repainted) to a screen, the operating system plays an important part. It takes a lot of responsibility for deciding how much information is needed from the application in order to render the window on the screen, and if it needs any information, it fires a `Paint` event.

Depending on the nature of the graphic, you can sometimes use the information in the `ClipRectangle` property to minimize the amount of work the `Paint` event handler must do. For example, suppose you have an application that creates a two-dimensional grid effect. There are many ways you could achieve this effect. One good way is to write the `Paint` event handler in such a way that a repaint requires minimal work within the event handler.

Let's look at an example to illustrate this. We're going to write some code that creates the graphic shown in [Figure 7-10](#). We'll also write some logging information to the console window, so we can track how much work the event handler is doing.



0,0	1,0	2,0	3,0	4,0
0,1	1,1	2,1	3,1	4,1
0,2	1,2	2,2	3,2	4,2
0,3	1,3	2,3	3,3	4,3
0,4	1,4	2,4	3,4	4,4
0,5	1,5	2,5	3,5	4,5
0,6	1,6	2,6	3,6	4,6
0,7	1,7	2,7	3,7	4,7
0,8	1,8	2,8	3,8	4,8
0,9	1,9	2,9	3,9	4,9
0,10	1,10	2,10	3,10	4,10
0,11	1,11	2,11	3,11	4,11

Figure 7-10: A grid

We'll build the grid effect by using a nested loop to write the rows and columns of rectangles, and the text strings within them. The nested loop is emphasized in the code that follows.

```
private void Form1_Paint (object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;

    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // Create essential objects for painting text strings
    SizeF sizeF = g.MeasureString("Test", this.Font);
    StringFormat sf = new StringFormat();
    sf.LineAlignment = StringAlignment.Center;

    // Set properties of the grid
    int cellHeight = (int)sizeF.Height + 4;
    int cellWidth = 80;
    int nbrColumns = 50;
    int nbrRows = 50;

    // Output general info to console
    Console.WriteLine("-----");
    Console.WriteLine("e.ClipRectangle = " + e.ClipRectangle);
    Console.WriteLine("The following cells need to be redrawn " +
        "(in whole or in part):");
```

```
// Draw the cells and the output to console
for (int row = 0; row < nbrRows; ++row)
{
    for (int col = 0; col < nbrColumns; ++col)
    {
        Point cellLocation = new Point(col * cellWidth,
                                         row * cellHeight);
        Rectangle cellRect = new Rectangle(cellLocation.X,
                                            cellLocation.Y,
                                            cellWidth, cellHeight);
        if (cellRect.IntersectsWith(e.ClipRectangle))
        {
            Console.WriteLine("Row:{0} Col:{1}", row, col);
            g.FillRectangle(Brushes.LightGray, cellRect);
            g.DrawRectangle(Pens.Black, cellRect);
            String s = String.Format("{0},{1}", col, row);
            g.DrawString(s, this.Font, Brushes.Black, cellRect, sf);
        }
        else
        {
            // Do nothing...
        }
    }
}
}
```

Here, the outside loop builds each row one at a time. For each row, the inside loop builds a number of rectangles side by side. But the important thing here is the `if . . . else` clause within the inside loop:

```
if (cellRect.IntersectsWith(e.ClipRectangle))
{
    ...
}
else
{
    // Do nothing...
}
```

This clause states that the event handler will build this rectangle *only* if its intersection with the `ClipRectangle` is not empty. In other words, it will not build the rectangle if the operating system isn't going to render at least part of that rectangle to the drawing surface.

To see the effect of this, just run the example. Without the `if...else` clause, the event handler would need to generate 50 rows and 50 columns—2,500 rectangles! But since the `if . . . else` clause compares the intended position of each rectangle with the `ClipRectangle`, the only rectangles generated by the code are the ones that will actually be painted to the visible drawing surface. The output in the console window shows exactly that:

```
e.ClipRectangle = {X=0,Y=0,Width=350,Height=200}
The following cells need to be redrawn (in whole or in part):
Row:0 Col:0
Row:0 Col:1
Row:0 Col:2
Row:0 Col:3
Row:0 Col:4
Row:1 Col:0
Row:1 Col:1
Row:1 Col:2
Row:1 Col:3
Row:1 Col:4
    <snipped...>
Row:11 Col:0
Row:11 Col:1
Row:11 Col:2
Row:11 Col:3
```

With the `ClipRectangle` at 350X200 pixels, the event handler needs to draw only the first 11 rows and 4 columns. The (nonempty) intersection of these rectangles with the `ClipRectangle` is the graphic that is returned to the operating system for rendering. The same applies if you resize the window, as you'll see if you try it out.

Forcing a Repaint of the Entire Window

Sometimes, the content of your window is such that, if the size of the window changes, you *want* to repaint the *entire* window. For example, suppose your window contains a bitmap that you have stretched to fill the client rectangle. This is what we did in an example in [Chapter 5](#), as shown in [Figure 7-11](#) (remember Rama the dog?).

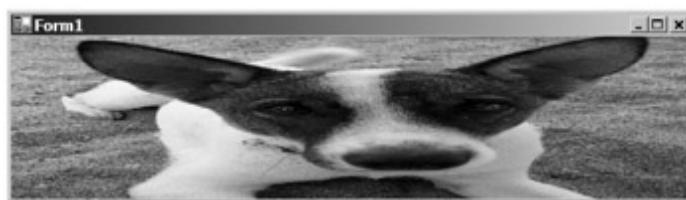


Figure 7-11: Stretched bitmap

In this case, it's not sufficient simply for the `Paint` event to supply a region containing a portion of the graphic, because any resize of this window will affect the entire client rectangle. In contrast, if you were not drawing a stretched image, resizing the window by making it bigger would mean that you would need to draw only the new area. Instead, you need to be able to force a resize to cause the *entire* client rectangle to be repainted.

You cannot overwrite the `EventArgs.ClipRectangle` property, because it's a read-only property. However, you can force a complete redraw by setting the `ResizeRedraw` property of the `Form` to true. Add a `Load` event to the form, and set the `ResizeRedraw` property there:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.ResizeRedraw = true;
}
```

This causes the `ClipRectangle` to be the same as the client rectangle every time the window is redrawn:

```
Clipping region painted: {X=0,Y=0,Width=250,Height=200}
Clipping region painted: {X=0,Y=0,Width=311,Height=328}
Clipping region painted: {X=0,Y=0,Width=433,Height=113}
Clipping region painted: {X=0,Y=0,Width=313,Height=303}
```

Previous

Next

Previous

Next

Clipping at the Pixel Level

As you've seen, one of the main purposes of the clipping algorithms implemented by GDI+ (and, indeed, by other drawing packages) is to support the behavior of overlapping windows on the computer screen. This fact is one that strongly influences the way that clipping behaves at the pixel level.

The reason for this is that on your raster-based computer screen, each window is composed of an array of *whole* pixels. Each edge of each window coincides with the edge of a row or column of pixels, and each edge of the client rectangle also coincides with the edge of a row or column of pixels. There is no anti-aliasing (of the sort described in [Chapter 2](#)) at the window's edge or at the client rectangle's edge.

What this means is, at the pixel level, your clipping region is also composed of *whole* pixels. It is not composed of half pixels, in any sense. In other words, the edge of a clipping region does not include part

pixels. Neither does it include pixels whose color is influenced by both the color of the region and the color of the neighboring thing outside the region.

To illustrate this point, we can compare the close-up results of a couple of examples. In both examples, we will apply anti-aliasing. In the first example, we'll take a pixel-level view of a clipping region drawn onto a drawing surface:

```
Graphics g = e.Graphics;
g.FillRectangle(Brushes.White, this.ClientRectangle);

g.SmoothingMode = SmoothingMode.AntiAlias;
Rectangle rect = new Rectangle(2, 2, 4, 4);
g.SetClip(rect);
g.FillRectangle(Brushes.Black, this.ClientRectangle);
```

[Figure 7-12](#) shows the result.

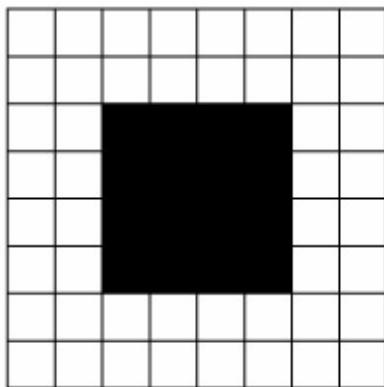


Figure 7-12: Clip rectangle— 2, 2, 4, 4 with anti-aliasing

In the next example, we'll look at a rectangle drawn onto the drawing surface with antialiasing turned on:

```
Graphics g = e.Graphics;
g.FillRectangle(Brushes.White, this.ClientRectangle);

g.SmoothingMode = SmoothingMode.AntiAlias;
Rectangle rect = new Rectangle(2, 2, 4, 4);
g.FillRectangle(Brushes.Black, rect);
```

The result of this example is shown in [Figure 7-13](#).

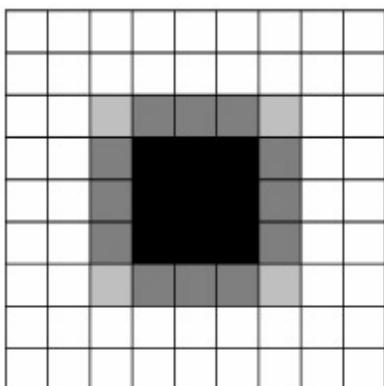


Figure 7-13: Drawn rectangle— 2, 2, 4, 4 with anti-aliasing

The effect of anti-aliasing in the second example is clear, and similar to the effects you saw in [Chapter 2](#). More important, the drawn rectangle consists of partially shaded pixels at the rectangle's edge (as described in [Chapter 2](#)). However, in the first example, the anti-aliasing setting has been ignored, and the region consists of *whole* pixels only.

 PreviousNext 

Using Invalidation

Broadly speaking, two sets of circumstances can cause your drawing surface to need to be redrawn: external effects and internal effects. We have discussed some external effects already. For example, if a user resizes the window, this is an action that is outside the control of the application—users can change the window size whenever they want. When it does happen, the operating system will detect the user's action and raise a `Paint` event, which is then handled by the application.

On the other hand, it is possible that things may happen *within* the application that cause you to want to redraw the graphic. For example, if the user performs an operation on the control that changes the state of the control, such as checking a check box within the control, you will want to be able to redraw the control to reflect the user's action.

In circumstances like this, you need to be able to *force* a `Paint` event from within the application.

Invalidation is a mechanism that allows you to programmatically force the .NET Framework to raise a `Paint` event. In effect, you use invalidation to tell GDI+ that a specified portion of the window or custom control is no longer valid and needs to be repainted.

Using State in Controls

Most custom controls involve a certain amount of *state*—internal data that is related to the control. For example, a control with a check box is likely to need to be able to keep a record of the current value of the check box. It would do so by storing the value in an internal member variable within the control. When the user checks or unchecks the check box, the control needs to update the internal variable (that is, update the state) to reflect this change.

You can maintain the state of a control in various ways. You might keep it in internal member variables, in a specially designed data structure, or even in a special state object.

No matter how you store your control's state internally, the fact remains that the state probably has a significant effect on the appearance of the control. So, if a user performs an action that changes the state of the control, you will probably need to take these changes into account and redraw the control accordingly. As mentioned, this means that you must be able to force a `Paint` event from within the application, and this is the purpose of invalidation.

The best way to react to an action like this is as follows:

1. *Change the state.* In the case of the check box, you might have a Boolean member variable that is set to `true` or `false`, and needs to be changed when the check box is clicked.
2. *Identify the region* of the drawing surface that is *affected by the change of state*. For example, if the user clicked on a check box, then you will need to change the appearance of the check box. Depending on the exact functionality of the control, you may need to update text or other regions of the drawing surface, too.
3. If the region identified in step 2 is currently visible, then *invalidate it*. To do this, use the `Invalidate` method. `Invalidate` is a method of the `Control` class. Because custom controls derive from the `Control` class, custom controls also have the `Invalidate` method. The invalidated region is passed as an argument to this method.
4. *Wait for the Paint event* to come through to your custom control. The clipping region will be set appropriately. Alternatively, force a `Paint` event by using the `Update` method.

This approach has several benefits to recommend it:

- It consolidates *almost all* painting code for the check box into a single method: the `Paint` event handler. As a consequence, your code is cleaner, more modular, and therefore more maintainable. This is arguably the most important benefit.
- In a complex custom control, you may have many event handlers that all need to be able to change the control's state in response to user actions and operating system actions, including mouse, keyboard, timer, and other event handlers. This approach allows you to standardize the way that these event handlers change the appearance of the custom control. They all use the same approach:

- This is the standard approach that developers follow when building custom controls. By following this pattern, you will ensure that your custom controls are easier for maintenance developers to understand. The structure of your control will not be a surprise to them.

Note The approach covered here works well for almost all events. The one exception to this rule is when you're implementing an operation that involves dragging something with the mouse. In this case, the dragging effect might be shown semitransparently on the screen. The nature of a drag event is different from most other events, because it is not a discrete event. If you go through the process of changing state, invalidating, and then waiting for the Paint event with every mouse movement, the dragging operation is likely to appear jerky and amateurish. The solution to this problem is to include the drawing code within the mouse movement event. By using bit block transfer (`BitBlt`), you can create a very sophisticated appearance for your drag-and-drop operations, similar to that of the file manager under Windows XP. You'll learn about this technique in [Chapter 14](#).

How Invalidation Works

You invalidate a window by calling the `Invalidate` method, which is implemented by the `Control` class and is thus available through the `Form` class (because the `Form` class derives from the `Control` class).

Of course, invalidation applies only to drawing surfaces that are rendered to the screen. The other two main types of drawing surfaces—printers and images—don't have any use for an invalidation mechanism. The print preview drawing surface, although rendered to the screen, really has the behavior of a printer, and it also doesn't use an invalidation mechanism.

The invalidation mechanism works as follows:

- Somewhere in your application, your code makes a decision that the contents of your window or custom control need to change. You then call the `Invalidate` method, which is overloaded. If you call the method with no arguments, the entire control or window is invalidated. Alternatively, you can pass a rectangle or a region to the `Invalidate` method, which will then invalidate just the area that you specify.
- At some time after you call the `Invalidate` method, GDI+ will send a `Paint` event to your window or custom control. You have no guarantee as to when this `Paint` event will be raised. The `Paint` event may not come through immediately. If, after invalidating, you continue to do your own processing, the `Paint` event will not come through until you relinquish control by returning from whatever event is currently executing. In addition, if you have invalidated several windows and controls, you cannot dictate the order in which your `Paint` events are raised.
- When the `Paint` event is raised, GDI+ sets the clipping region to the area or region that you invalidated. If you invalidate several rectangles or regions at once, GDI+ may coalesce these into one region and raise a single `Paint` event.
- If you want to force the raising of the `Paint` event for your invalidation, you can do so by calling the `Form.Update` method, which is also inherited from the `Control` class. If you call the `Update` method, a `Paint` event will occur before the `Update` method returns to the method that called it.
- In many cases, the clipping region that comes with the `Paint` event is the same rectangle or region as that passed to the `Invalidate` method, unless GDI+ coalesces multiple invalidations into one.

Note that invalidation uses the same coordinate system behavior as clipping, and hence invalidation occurs at the granularity of pixels, not subpixels. This is not really surprising, since invalidation is dependent on clipping capabilities.

Changing a Control's State

Let's look at an example that uses invalidation. In this example, the "state" will consist of the value of a single variable, `strColor`. To enable this value to be accessible to various event handlers, but not outside of our class, we declare it as a private member variable:

```
public partial class Form1 : Form
```

```
{  
    private string strColor;
```

The application's state will need to be initialized. To do this, we assign the value red to the string strColor within the form's InitializeComponent routine (this method resides in the partial class Form1):

```
private void InitializeComponent()  
{  
    ...  
    strColor = "red";  
}
```

Now, our application will need two event handlers. First, we'll need a Paint event handler to paint to the drawing surface. We'll also write a Click event handler.

The Paint Event Handler

Let's write the Paint event handler first. We'll use it to draw a traffic-control type graphic, which displays one of three colored "lights"—red, yellow, or green—depending on the value of strColor. When we first fire up the application, the red light will show.

The code in the Paint event handler itself is nothing unusual. We use a C# switch statement to detect the value of strColor and paint the correct light. We also write the dimensions of the ClipRectangle to the console window, because that will be of interest to us in a moment:

```
private void Form1_Paint(object sender,  
    System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, this.ClientRectangle);  
  
    // Report dimensions of clipping rectangle to console  
    Console.WriteLine("e.ClipRectangle = " + e.ClipRectangle);  
  
    // Draw outline  
    g.FillRectangle(Brushes.Black, 10, 10, 50, 150);  
    g.DrawEllipse(Pens.White, 15, 15, 40, 40);  
    g.DrawEllipse(Pens.White, 15, 60, 40, 40);  
    g.DrawEllipse(Pens.White, 15, 105, 40, 40);  
  
    // Use application state (strColor) to draw  
    // exactly one of the three lights  
    switch(strColor)  
    {  
        case "red":  
            g.FillEllipse(Brushes.Red, 15, 15, 40, 40);  
            break;  
        case "yellow":  
            g.FillEllipse(Brushes.Yellow, 15, 60, 40, 40);  
            break;  
        case "green":  
            g.FillEllipse(Brushes.Green, 15, 105, 40, 40);  
            break;  
        default:  
            g.FillEllipse(Brushes.Red, 15, 15, 40, 40);  
            break;  
    }  
}
```

[Figure 7-14](#) shows what this looks like when we start the application. The light at the top is colored red.

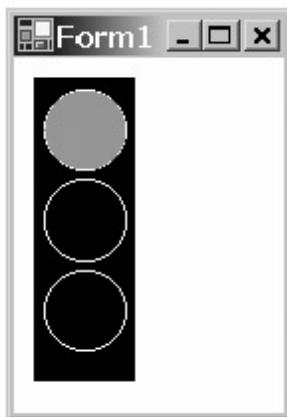


Figure 7-14: A stoplight control

The Click Event Handler

Now, the `Click` event will force a change to the state of the application. The `Click` event handler will check the current value of `strColor` and change it to something different.

```
private void Form1_Click(object sender, System.EventArgs e)
{
    // Change application state by altering value of strColor
    switch(strColor)
    {
        case "red":
            strColor="yellow";
            break;
        case "yellow":
            strColor="green";
            break;
        default:
            strColor="red";
            break;
    }
}
```

Of course, once we've changed the state (by changing the value of `strColor`), the display shown on the drawing surface no longer corresponds to the application's state. For example, when the `Click` event causes the value of `strColor` to change from `red` to `yellow`, the drawing surface itself needs to be redrawn to reflect the change: we need to show the yellow light instead of the red one.

To do this, we invalidate the region of the drawing surface that needs redrawing. A `Paint` event will be fired sometime after that, but to force the `Paint` event, we follow the `Invalidate` method call with a call to the `Update` method:

```
// Invalidate the region containing the traffic lights
this.Invalidate(new Rectangle(10, 10, 50, 150));
}
```

Each time we click the window, the state changes. The rectangular region is marked as invalid, the `Paint` event fires, and the window is repainted so that it reflects the state of the control. [Figure 7-15](#) shows the three states of the stoplight.

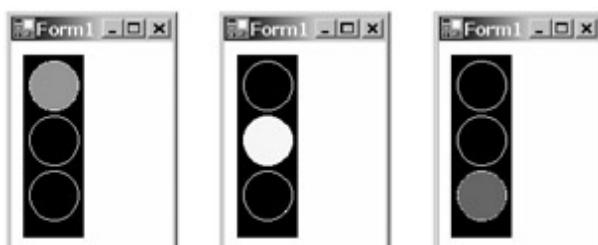


Figure 7-15: Three states of the stoplight

The Clipping Rectangle

The final behavior to note here is that the first time the window is drawn, the clipping rectangle is the same size as the client rectangle. But whenever we change the state, we invalidate only the small rectangle that contains the traffic light graphic:

```
this.Invalidate(new Rectangle(10, 10, 50, 150));
```

Therefore, when the `Paint` event runs, the clipping rectangle is set to be the invalidated region, and this is the *only* portion of the window that is redrawn. You can see this in the output to the console window:

```
e.ClipRectangle = {X=0,Y=0,Width=120,Height=180}
e.ClipRectangle = {X=10,Y=10,Width=50,Height=150}
e.ClipRectangle = {X=10,Y=10,Width=50,Height=150}
e.ClipRectangle = {X=10,Y=10,Width=50,Height=150}
```

Previous

Next

Previous

Next

A Few Clipping Tips

Some applications have very simple clipping algorithms. They may need to set a clipping rectangle or two every now and then. Other applications are much more elaborate. In my experience, if the application or custom control being developed needs a more elaborate clipping scheme, using the following techniques leads to a more maintainable application:

- Store the clipping region as part of the state of the window, such as in a private member variable, rather than in the `Paint` event. This allows any event handler or helper function to access and change the clipping region if necessary. It also helps future maintenance programmers to identify the clipping algorithm. This technique means that you would tend to use only the `Replace` member of the `CombineMode` enumeration.
- When you need to modify the clipping region, modify the state variable, and then explicitly replace the clipping region before you draw.
- When you create a complicated control with drawing operations spread over many methods, have each method set its clipping region before it starts to draw. In other words, don't allow the method to make assumptions about the value of the clipping region stored in state. Setting the clipping region is a cheap operation; it takes very little processing time. Even if you unnecessarily set the same clipping region many times, the resulting clarity of code more than makes up for the slight increase in processing time.

Previous

Next

Previous

Next

Summary

[Chapter 6](#) developed the idea of a region. In this chapter, we focused on the clipping region, which is the region of the drawing surface in which the effects of drawing actions will actually be seen. We explored two important applications of the clipping region.

First, we looked at how you can apply the clipping region like a stencil, and hence create a cropping effect. In this application, you control the clipping region programmatically, using methods and properties of the `Graphics` object (such as the `Clip` property and the `SetClip` method). You can set the clipping region using a `Rectangle`, `RectangleF`, `Graphics`, `GraphicsPath`, or `Region` object. You can also compose the clipping region using standard set algebra operations: `Complement`, `Exclude`, `Intersect`, `Union`, or `Xor`.

Second, you saw that when a `Paint` event is raised, the event handler doesn't *always* need to regenerate the entire drawing surface. The event handler is passed information about how much of the drawing surface it needs to redraw, in the form of the `PaintEventArgs.ClipRectangle`. Within the event handler, you can ignore this information if you choose; but sometimes, it's possible to use the

ClipRectangle information to minimize the amount of work that the event handler needs to do, and therefore improve the efficiency of the event handler.

The ClipRectangle is an important tool. We studied the dimensions of the ClipRectangle requested in a number of different circumstances, particularly in situations where the control is being rendered to a window on a screen:

- Sometimes, the ClipRectangle is the same as the client rectangle. For example, this happens when the application first draws the drawing surface. It must generate enough of the graphic to fill the client rectangle.
- If the user resizes the window, the operating system will compare the old window size and the new window size. If the new size is longer and/or wider than the old size, the resulting client rectangle will expose regions of the drawing surface that weren't previously exposed. Since the operating system doesn't know what to render in those regions, it fires a Paint event on the application. It passes a ClipRectangle that tells the event handler which region of the drawing surface needs to be regenerated. The ClipRectangle will be the smallest possible rectangle that contains the missing information.
- If the user obscures part of the window with another window, and then reveals it again, the operating system needs to redraw the section that was obscured. Again, it fires a Paint event, passing a ClipRectangle that corresponds to the obscured portion of the drawing surface (which is the only part that needs to be repainted).
- If the state of your application changes (for example, as a result of a user's mouse or keyboard event), and this needs to be reflected in the appearance of the control, you need a way to have the control repainted. You can force a Paint event programmatically from within the application by using invalidation. You can invalidate the entire drawing surface or just the region that is affected by the change of state. The dimensions of the invalidated region are passed back into the Paint event via the ClipRectangle, and so only the invalidated region is repainted.

We also took a brief look at how to use state in controls. It's best to hold state in member variables in your class. The result is more modularized application code in which the drawing code is isolated in the Paint event handler (with one exception: the mousemove event), and therefore the code is easier to maintain.

In the [next chapter](#), we're going to move away from the subject of regions and tackle another important tool: transformations.

 Previous

Next 

[Previous](#)[Next](#)

Chapter 8: Transformations

Overview

In this chapter, we'll look at the way in which GDI+ implements an advanced and important graphics tool: the transformation. Transformations are a facility that you can use to apply a conversion (or a set of conversions) to all drawing operations. When you have a given transformation in effect, this affects all drawing operations—drawing text, drawing lines, and drawing and filling shapes. Transformations are also applied to graphics paths and regions.

For example, you might use a transformation in an application where users need to zoom in and out of a technical drawing. You can achieve this effect by applying a scaling transformation before drawing. As the user zooms in and out, you can change the scaling factor that is applied when setting up the transformation.

Although transformations are not often used in custom control development, they can be very useful in certain circumstances. In fact, in [Chapter 13](#), you'll see how to use transformations in combination with other techniques to build a scrolling control that has more capabilities than the stock .NET Windows Forms scrolling controls.

This chapter covers the following topics:

- Coordinate systems and conversions
- How to implement scaling, translation, rotation, and shearing transformations using the methods available in GDI+
- When custom control developers might find transformations useful

First, let's identify the different types of transformations and the practical results they give.

[Previous](#)[Next](#)[Previous](#)[Next](#)

Types of Transformations in GDI+

In GDI+, when you apply a transformation to a drawing surface, it applies to all subsequent drawing operations. As each drawing operation executes, GDI+ alters the results of the drawing operation by applying the current transformation to it.

To understand the types of GDI+ transformations, we'll start with the normal (untransformed) letter A drawn in a window, as shown in [Figure 8-1](#).

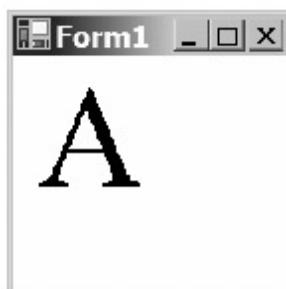


Figure 8-1: Untransformed letter A

In GDI+, there are four basic types of transformations:

- **Scaling transformation:** This type of transformation changes the size of the result. So if you apply a

scaling transformation that increases the size of the subsequent drawing operations, the result when you draw the letter A appears to be larger, as shown in [Figure 8-2](#).

- **Translation transformation:** This type of transformation has the effect of shifting the results of subsequent operations a specified distance left or right, and up or down. So, if you apply a translation that shifts down and to the right to your drawing surface, and then draw the letter A, the result might look like [Figure 8-3](#).
- **Rotation transformation:** This type of transformation has the effect of rotating elements through a specified angle about a specified point. [Figure 8-4](#) shows an example (although, as you'll see later, the example isn't quite as straightforward as it looks).
- **Shearing transformation:** This type of transformation reminds me of how a cartoonist conveys an image of a car moving fast by drawing the car leaning backwards. Mathematically, a shear is a transformation that turns a rectangle into a parallelogram. [Figure 8-5](#) shows an example with the same letter A.



Figure 8-2: Scaling transformation

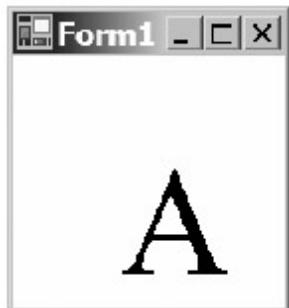


Figure 8-3: Translation transformation

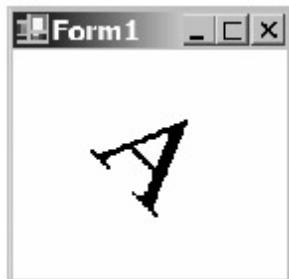


Figure 8-4: Rotation transformation



Figure 8-5: Shearing transformation

These are the four basic types of transformations that we'll work with in this chapter. In a moment, we'll dig into the specifics of each operation, but first, we need to return to the subject of coordinates, to understand the different coordinate systems in GDI+.

Previous

Next

Previous

Next

Coordinate Systems and Transformations

We discussed coordinate systems in some detail back in [Chapter 2](#), when we first looked at drawing surfaces. You've seen the influence of coordinate systems in the treatment of texts, fonts, and images, and it also looms large when you're printing, as you'll learn in [Chapter 9](#).

In fact, GDI+ has three different coordinate systems that you must know about:

- **World coordinates:** The world coordinate system is the one that you reference in your application. Whenever you pass coordinates to any drawing function, you express those coordinates in the world coordinate system. Any transformations that you apply transform your coordinates from world coordinates to page coordinates.
- **Page coordinates:** The configuration of the page coordinate system determines the units in which you draw. Its origin is at the upper-left corner of the client area. By default, you express page coordinates in terms of pixels. If you change the unit of measure for the page from pixels to, say, inches, this conversion takes place to give device coordinates.
- **Device coordinates:** The device coordinate system is based on pixel coordinates, which, like page coordinates, are relative to the upper-left corner of the client area (or the upper-left corner of the printable area of the page).

So, when GDI+ outputs to a device (like a screen) there are two coordinate conversions that take place: a *world transformation*, which converts from world coordinates to page coordinates, and a *page transformation*, which converts from page coordinates to device coordinates.

You might be drawing a very intricate diagram that you expect to look acceptable when it's drawn very small, perhaps using anti-aliasing, and you also want it to look good when you scale up the drawing, so that it fills a large window. To facilitate both of these situations, coordinates are often expressed with floating-point precision in the world coordinate system.

By default, the page coordinate system unit of measure is `Pixel`, which is the most useful unit for making custom controls, as noted in [Chapter 2](#). However, for other applications, such as writing code that draws engineering diagrams, you may want to express coordinates in units other than pixels. If you set the page coordinate system to another value—such as `Inch`, `Millimeter`, or `Document`—your coordinates are then expressed in the associated unit of measure (inch, millimeter, or 1/300 inch, respectively).

The *device coordinate system* is always pixel-based. This is related to the fact that all drawing surfaces in GDI+ are raster-based.

This chapter focuses specifically on the world transformation, which is the mechanism by which you can carry out translations, scaling operations, rotations, and shearing.

Note The page transformation is specified using the `PageUnit` and `PageScale` properties of the `Graphics` class. The `PageUnit` property was introduced in our discussion of texts and fonts, and it will be discussed again in more detail in [Chapter 9](#).

World Transformations

When a world transformation is set, it applies to all items that are subsequently drawn on the surface. For a large part of this chapter, we'll employ certain convenience methods made available by the `Graphics` class, which help you carry out the most common transformations. It is worth bearing in mind though that, under the hood, what you are really doing is manipulating a `Matrix` object. When you call the methods that set up the transformation, these methods create and initialize a `Matrix` object, which drives the actual transformation. These methods are called *convenience methods* because it is far easier to call them than to directly set up the `Matrix` object. [Table 8-1](#) shows the convenience methods we'll use in this chapter.

Table 8-1: Transform Methods

Method	Description
<code>ScaleTransform</code>	Applies scaling to the world transformation.
<code>TranslateTransform</code>	Applies a translation (horizontal and/or vertical shift) to the world transformation.
<code>RotateTransform</code>	Applies a rotation about the point (0, 0), specified in degrees, to the world transformation.
<code>MultiplyTransform</code>	Appends a transformation <code>Matrix</code> object to the world transformation. This means that if you had an existing transformation in place, using the <code>MultiplyTransform</code> method appends the specified transformation to the existing transformations.
<code>ResetTransform</code>	Resets the transformation to the identity matrix. This means that no transformations are in effect. This basically clears all existing world transformations.

A `Matrix` object defines what is called an *affine* transformation, which is a transformation that preserves colinearity, meaning that parallel lines always stay parallel after the transformation. In GDI+, no matter which type of transformation you set up—scaling, translation, shearing, and so on—it is an affine transformation. We don't have any means for doing other kinds of transformations.

Scaling Transformations

A scaling transformation simply allows you to apply a horizontal scaling factor and a vertical scaling factor. For example, if the scaling factor applied is 1.0, then no scaling takes place. If the scaling factor is 0.5, then subsequent elements are scaled to half size. If the scaling factor is 2.0, the elements are drawn at double size.

Here's a simple `Paint` event handler that contains code to illustrate a scaling transformation:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    // First, draw a nonscaled rectangle and circle
    g.DrawRectangle(Pens.Black, 10, 10, 50, 50);
    g.DrawEllipse(Pens.Black, 10, 10, 10, 10);

    // Now apply the scaling transformation
    // This will scale subsequent operations by 2x horizontally
    // and 3x vertically
    g.ScaleTransform(2.0f, 3.0f);
```

```
// Now draw the same rectangle and circle,  
// but with the scaling transformation  
g.DrawRectangle(Pens.Black, 10, 10, 50, 50);  
g.DrawEllipse(Pens.Black, 10, 10, 10, 10);  
}
```

When you run this example, the results look like [Figure 8-6](#).

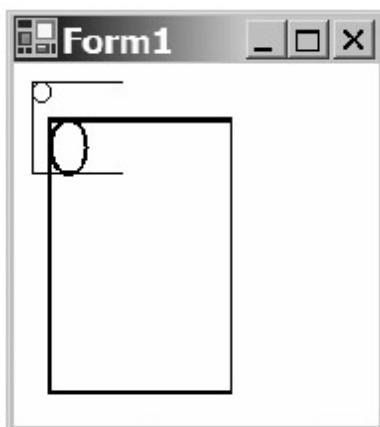


Figure 8-6: Using ScaleTransform

This example includes four drawing operations, separated by one transformation method call. The first two drawing operations draw a square and a circle to the drawing surface. At the time these two drawing operations are called, there is no transformation specified, so GDI+ adopts the default world transformation. In other words, it maps the world coordinate system directly onto the page coordinate system, mapping (0, 0) in the world coordinate system to (0, 0) in the page coordinate system and applying no scaling, translation, or shear transformation effects. The result is the "natural" square and circle you see in [Figure 8-6](#).

Then we apply the `ScaleTransform` method and draw the same objects again. This time, the transformation has the effect of scaling the dimensions of the rectangle and circle (doubling in the `X` direction and tripling in the `Y` direction). The circle becomes an oval after the transformation. Note that it has also scaled the width of the pen used to draw the shapes.

You can also use the `ScaleTransform` method to yield reflections. You'll see how this works in the ["Transformations and Matrices"](#) section later in this chapter.

Translation Transformations

In the development of custom controls, the translation transformation is probably the most common type. Edit controls, grid controls, virtual spaces, and many other types of custom controls have the potential to use it.

With a translation transformation, you apply a shift in the `X` and/or `Y` directions. Positive values shift the resulting graphics operations down and to the right. For example, if you were drawing a line starting at point (10, 10), and you applied a translation transformation of `X = 5, Y = 10`, then the line would actually be drawn at pixel (15, 20).

Consider the following code:

```
private void Form1_Paint(object sender,  
    System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, this.ClientRectangle);  
    Font f = new Font("Times New Roman", 24);  
  
    // Draw text to the surface  
    g.DrawString("Translation", f, Brushes.Black, 0, 0);
```

```
// Translate the text 150 pixels horizontally and 75 vertically
g.TranslateTransform(150, 75);

// Draw the translated text
g.DrawString("Translation", f, Brushes.Black, 0, 0);
}
```

This results in a window with the word *Translation* written on it twice in different places, as shown in [Figure 8-7](#), which also has some annotations showing how the translation works.

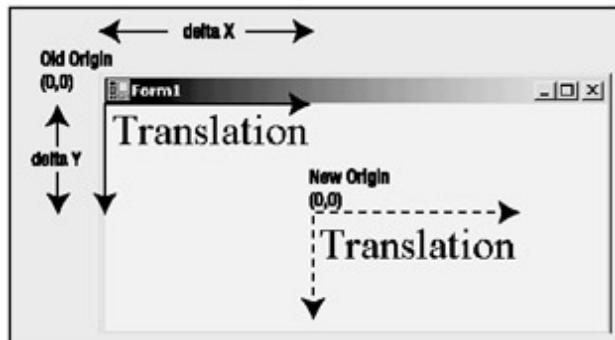


Figure 8-7: Using TranslateTransform

Mathematically speaking, the translation has moved the origin to which the drawing operation is related. In effect, applying a translation transformation means that the origin $(0, 0)$ of the drawing surface has been moved from the top-left corner of the client rectangle, by a distance $\text{delta } X = 150$ pixels to the right and $\text{delta } Y = 75$ pixels down. The drawing operation is oriented along axes that are parallel to the original axes (because we have applied only a translation, not a rotation or shear).

Remember that GDI+ applies the current world transformation to each drawing operation in turn, converting world coordinates to page coordinates. Thus, for the *first* drawing operation here (where there is no world transformation explicitly defined), GDI+ adopts the default behavior—mapping the world coordinate system directly onto the page coordinate system—and hence places the word *Translation* faithfully near the upper-left corner of the client area.

Then the following call applies a world transformation of a translation (by 150 pixels in the horizontal and 75 pixels in the vertical direction):

```
g.TranslateTransform(150, 75);
```

So, the page coordinates for the second drawing operation are modified (from their origin of the upper-left corner of the client area) by this amount, and the second `DrawString` call yields text away from the corner. In other words, we can say:

$$X(\text{page coordinate}) = X(\text{world coordinate}) + \text{delta } X$$

$$Y(\text{page coordinate}) = Y(\text{world coordinate}) + \text{delta } Y$$

The primary reason for using the translation transformation is when you want to create a virtual space. By applying a translation transformation in the appropriate way, your drawing code does not need to be aware that it is drawing into a virtual space. This simplifies your drawing code.

Cumulative Transformations and Noncumulative Transformations

If you apply multiple transformations, they have a cumulative effect. Replace the code in the `Paint` event, as follows:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);

    for (int i = 1; i <= 5; ++i)
    {
        g.TranslateTransform(150, 75);
        g.DrawString("Translation", f, Brushes.Black, 0, 0);
    }
}
```

```
// First, draw a rectangle with the current transformation  
g.DrawRectangle(Pens.Black, 10, 10, 30, 50);  
  
// Add a translation to the existing transformation  
g.TranslateTransform(2, 10);  
}  
}
```

When you run this example, it generates a series of identically sized rectangles in different positions, as shown in [Figure 8-8](#).

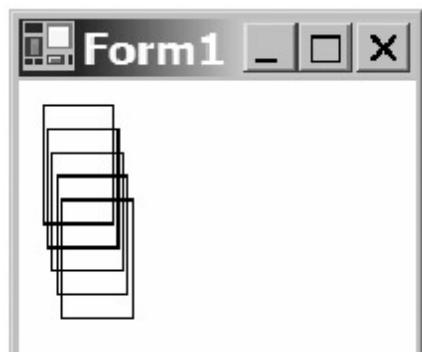


Figure 8-8: Cumulative transformations

If you want to apply transformations in such a way that they are not cumulative, you can call the `ResetTransform` method:

```
private void Form1_Paint(object sender,  
    System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, ClientRectangle);  
  
    // First, draw a rectangle with no translation  
    g.DrawEllipse(Pens.Black, 20, 20, 30, 50);  
  
    // Translate by -15 pixels in horizontal direction  
    g.TranslateTransform(-15, 0);  
    g.DrawEllipse(Pens.Black, 20, 20, 30, 50);  
  
    // Reset the transformation  
    // Translate by 30 pixels in vertical direction  
    g.ResetTransform();  
  
    g.TranslateTransform(0, 30);  
    g.DrawEllipse(Pens.Black, 20, 20, 30, 50);  
}
```

This has the results that you would expect, as shown in [Figure 8-9](#).

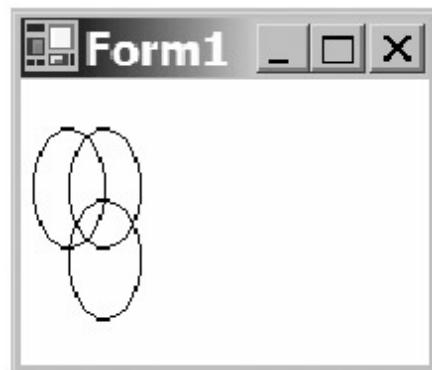


Figure 8-9: Using ResetTransform

The first ellipse is drawn faithfully at (20, 20), while the second is translated a distance 15 pixels left. Then we reset the world transformation to the default, before applying a downward translation. Thus, the two transformations in this example are not cumulative.

Rotation Transformations

With the rotation transformation, you apply a rotation in the *clockwise* direction, specified in degrees. The *center of rotation* is the *current* origin of the coordinate system.

To examine this, run the following code:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Font f = new Font("Times New Roman", 24);

    // Draw text to the surface
    g.DrawString("Rotation", f, Brushes.Black, 0, 0);

    // Rotate the text through 45 degrees
    g.RotateTransform(45);

    // Draw the rotated text
    g.DrawString("Rotation", f, Brushes.Black, 0, 0);
}
```

[Figure 8-10](#) shows the results, with some annotations showing how the rotation works.

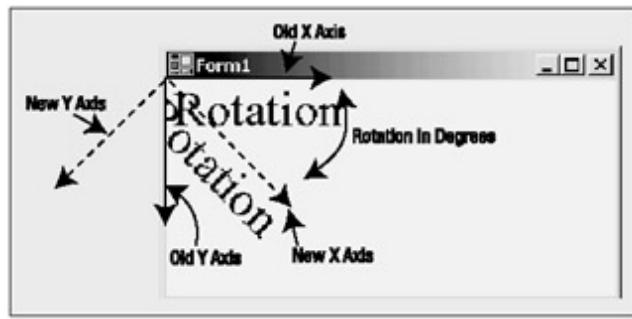


Figure 8-10: Using RotateTransform

Here, the effect of the transformation is to rotate the drawing surface (axes and all) against which the drawing operation is referenced.

Since rotation transformations rotate around the *current* origin (which, by default, is the upper-left corner), in this example, the rotated `DrawString` method draws a text string that is only partially visible on the client rectangle. Yet, in the rotation example shown earlier in [Figure 8-4](#), the letter A was rotated about a point other than the origin, in such a way that it didn't get rotated out of the window. To achieve this, I cheated a little. In fact, the letter A in [Figure 8-4](#) was both translated *and* rotated. This brings us neatly to the topic of combining different transformations.

Composite Transformations

As you may expect, there is no technical problem with carrying out any number of transformations one after another, in such a way that they accumulate into one (potentially very complex) world transformation. However, one point to note is that transformations are not commutative (to use the mathematical term). This means that the order in which you apply two transformations is important. For example, if you applied a translation and then a rotation, the result will not be the same as it would if you first applied the rotation and then the translation.

To see how this works, let's start with an X-axis translation followed by a 45-degree clockwise rotation:

```

private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Font f = new Font("Times New Roman", 24);

    // Draw text to the surface
    g.DrawString("Translate then Rotate", f, Brushes.Black, 0, 0);

    // Translate the text 150 pixels horizontally
    g.TranslateTransform(150, 0);

    // Rotate the text 45 degrees clockwise
    g.RotateTransform(45);

    // Draw the transformed text
    g.DrawString("Translate then Rotate ", f, Brushes.Black, 0, 0);
}

```

The origin of the drawing surface is shifted horizontally, and then the axes against which the operation is oriented are rotated, as shown in [Figure 8-11](#).

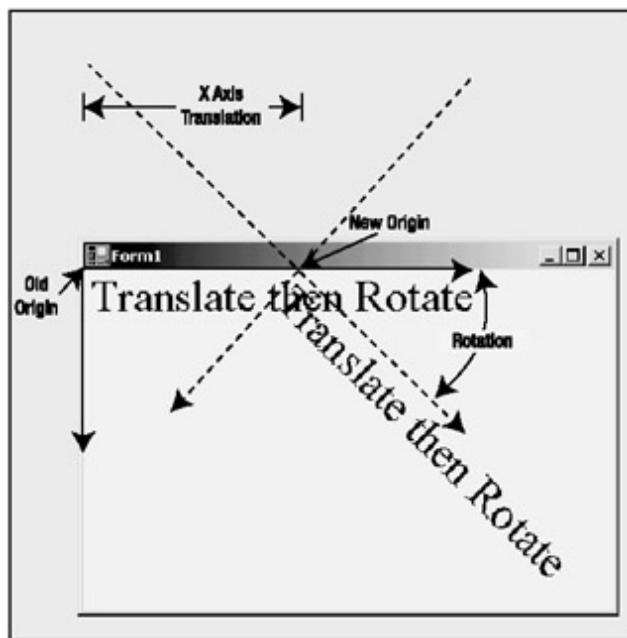


Figure 8-11: Translate, then rotate

Now let's perform the same transformations in the reverse order, by reversing the order of the transformation calls:

```

// Draw text to the surface
g.DrawString("Rotate then Translate", f, Brushes.Black, 0, 0);

// Rotate the text 45 degrees clockwise
g.RotateTransform(45);

// Translate the text 150 pixels horizontally
g.TranslateTransform(150, 0);

// Draw the transformed text
g.DrawString("Rotate then Translate ", f, Brushes.Black, 0, 0);

```

As shown in [Figure 8-12](#), this time the translation takes place along a rotated axis.

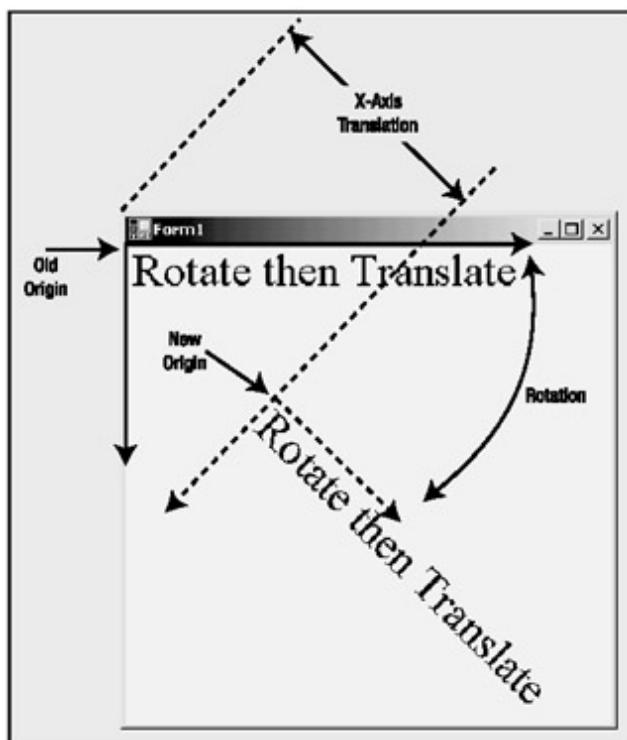


Figure 8-12: Rotate, then translate

The result is clearly different, demonstrating that you need to be aware of the order in which you apply transformations when you're accumulating transformations.

Just to round off this section, let's produce a small example that may trigger some thoughts about custom controls. Modify the Load event of a project as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.ResizeRedraw = true;
}
```

Add a Paint event, and modify it like so:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Font f = new Font("Times New Roman", 16);
    for (float angle = 0; angle < 360; angle += 45)
    {
        g.ResetTransform();
        g.TranslateTransform(ClientRectangle.Width/2,
            ClientRectangle.Height/2);
        g.RotateTransform(angle);
        g.DrawString("Hello, World", f, Brushes.Black, 50, 0);
    }
}
```

Each time the code in the loop is executed, the transformation is reset, and a horizontal and vertical translation are performed, followed by a rotation. When you run the example, you should see the results shown in [Figure 8-13](#).

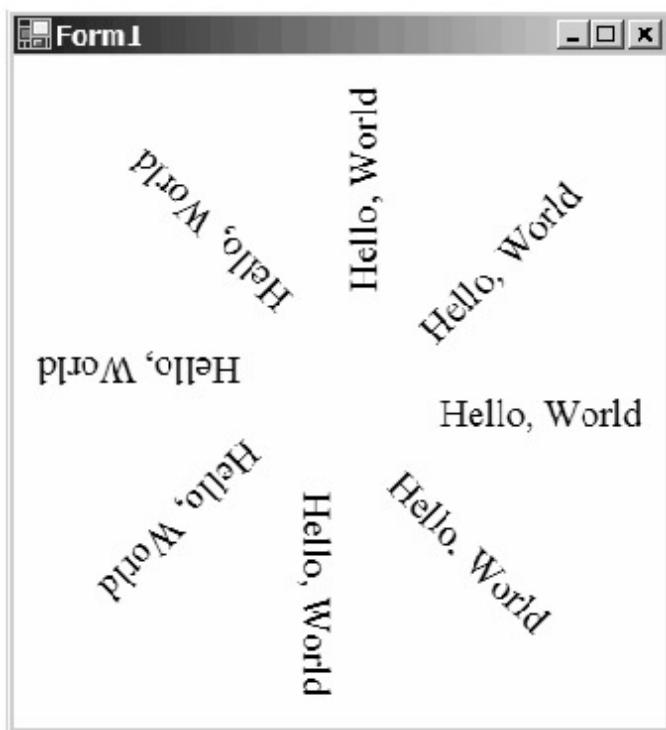


Figure 8-13: Rotated text

Transformations and Matrices

As noted earlier in the chapter, the convenience methods we've just been playing with obscure the use of the `Matrix` class. It is worth taking a little time to elaborate on the subject of how matrices are used in effecting transformations. Of course, if you're happy to use the convenience methods and/or you don't feel mathematically adventurous today, feel free to skip this section. But, as always, it's best to have some idea of what's going on behind the scenes. Also, this will help you to understand the reflection and shearing transformations, discussed in the next sections.

As you saw earlier, under translations for the X axis, the world transformation is:

$$X(\text{page coordinate}) = X(\text{world coordinate}) + \text{delta } X$$

Similarly, a scaling operation can be thought of like this:

$$X(\text{page coordinate}) = X(\text{world coordinate}) \times \text{scaling factor}$$

Rotations are more complex:

$$X(\text{page coordinate}) = X(\text{world coordinate}) \times \cosine(\text{angle}) + Y(\text{world coordinate}) \times \sin(\text{angle})$$

$$Y(\text{page coordinate}) = -X(\text{world coordinate}) \times \sin(\text{angle}) + Y(\text{world coordinate}) \times \cos(\text{angle})$$

Cumulative transformations obviously involve working through these formulas and combining terms. Quite clearly, this can get very complex very quickly, and matrix algebra is the easiest way to handle the situation.

Leaving out the mathematical derivation of matrix formation, we can represent the situation using the following expression. It shows how to calculate the final (transformed) X and Y coordinates of a point from its original coordinates, by applying matrix multiplication:

$$\begin{bmatrix} X_T & Y_T & 1 \end{bmatrix} = \begin{bmatrix} X_0 & Y_0 & 1 \end{bmatrix} \begin{bmatrix} L_1 & L_2 & 0 \\ L_3 & L_4 & 0 \\ T_1 & T_2 & 1 \end{bmatrix}$$

Here, X_0 and Y_0 represent the original coordinates, the 3x3 matrix contains all the information necessary for carrying out a world transformation, and X_T and Y_T are the page coordinates after transformation.

Note I've used matrix multiplication here. In matrix multiplication, the formula for calculating X_T in this expression is $X_T = (X_0XL_1) + (XXL_3) + (X_0XT_1)$. Y_T is calculated using a similar formula involving the second column of the matrix, and the third coordinate in the result 1 is calculated using the third column of the matrix.

Within the matrix, the L_i variables represent the linear part of the transformation (incorporate the information necessary for scaling and rotation), and the T_i variables contain any information required for a translation. Note that the 1 in the 3X1 matrix is a dummy coordinate, which enables the translation component to be included in one matrix. Since in GDI+, we are interested in only affine transformations, the third column contains the numbers [0 0 1].

As you saw in [Chapter 5](#), an identity matrix (which represents a transformation that has no effect) has ones down the leading diagonal and zeros everywhere else, as shown in [Figure 8-14](#).

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 8-14: The identity matrix

Now, when a new `Graphics` class is used, the world transformation in effect applies a transformation using the identity matrix. In other words, the page coordinates and the world coordinates match. Moreover, the `ResetTransform` method resets the transformation to the identity matrix, and thus clears all operative world transformations.

Indeed, the `Transform` property of the `Graphics` class is a `Matrix` object and is set by using the convenience methods we've discussed.

Note You can set the `Transform` property of the `Graphics` class directly. If you're interested, see the documentation for the `Graphics` class.

Continuing on this excursion into the topic of matrices, let's consider what happens if you use negative values in a scaling operation. The title of the [next section](#) might give you a clue.

Reflection Transformations

As noted earlier in the chapter, the `ScaleTransform` method can also be used for reflections. For example, if you were using GDI+ to build a child's drawing program, you might want the ability to draw flowers both forwards and backwards, for variety. A reflection transformation would suit your purposes here just fine.

Consider the following code:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, this.ClientRectangle);
    Font f = new Font("Times New Roman", 24);

    // Translate the text 175 pixels horizontally and 50 vertically
    // and draw text to the surface
```

```

g.TranslateTransform(175, 50);
g.DrawString("Reflection", f, Brushes.Black, 0, 0);

// Scale the text using a negative value
g.ScaleTransform(-1, 1);

// Draw the translated text
g.DrawString("Reflection", f, Brushes.Black, 0, 0);
}

```

The translation is carried out for convenience here, and it translates the origin to the point (175, 50). In particular, note that the translation moves the Y axis 175 pixels to the right of its original position.

The transformation matrix for the *scaling* is as shown in [Figure 8-15](#).

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 8-15: Reflection transformation matrix

Thus, our transformation can be represented by the following formula:

$$\begin{bmatrix} X_r & Y_r & 1 \end{bmatrix} = \begin{bmatrix} X_0 & Y_0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -X_0 & Y_0 & 1 \end{bmatrix}$$

So, while the Y-axis position remains constant, the X-axis values are negated, and thus the transformed text is written from right to left. The effect is a reflection in the Y axis, as shown in [Figure 8-16](#).

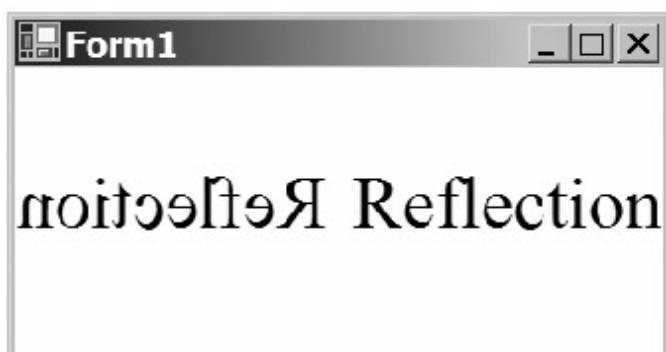


Figure 8-16: Reflected text

Shearing Transformations

When an airplane encounters wind shear, it is descending or ascending through an altitude where the wind speed above the airplane is different from that below it. In this situation, the airplane experiences a radical increase or decrease in lift, and it gets a big bump or drop.

A shearing transformation is one where the "wind" at the top of the drawing surface is blowing the pixels at a different rate from the "wind" at the bottom of the drawing surface. You get an effect where the graphics that you draw appear to lean forward or backward.

This transformation is not a common one, so there is no convenience method in the `Graphics` class to make it easy to apply. However, there is a convenience method in the `Matrix` class. This class is in the `System.Drawing.Drawing2D` namespace, so when trying this example, the first thing to do is add this to your file:

Then modify the Paint event, as follows:

```
private void Form1_Paint(object sender,  
    System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.White, ClientRectangle);  
  
    Matrix m = new Matrix();  
  
    // The following method call sets up the shear in the matrix.  
    // The first argument is the shear factor in the X direction.  
    // The second argument is the shear factor in the Y direction.  
    m.Shear(.6f, 0);  
    g.DrawRectangle(Pens.Black, 10, 10, 50, 50);  
    g.MultiplyTransform(m);  
    g.DrawRectangle(Pens.Black, 70, 10, 50, 50);  
}
```

Note The final DrawRectangle call in the example merely draws the rectangle in a different position for convenience.

When you run this example, the output looks like [Figure 8-17](#).

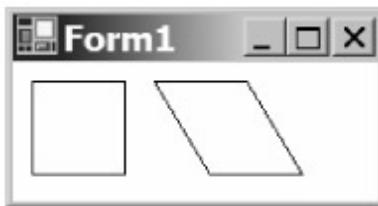


Figure 8-17: Shear transformation

The Shear method effectively sets up a matrix of the form shown in [Figure 8-18](#).

$$\begin{bmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 8-18: Shear transformation matrix

This gives a transformation of the form:

$$\begin{bmatrix} X_r & Y_r & 1 \end{bmatrix} = \begin{bmatrix} X_0 & Y_0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (X_0 + 0.6Y_0) & Y_0 & 1 \end{bmatrix}$$

Thus the X coordinate of a point after a shear transformation is dependent on its original X coordinate and also on its original distance from the Y axis—the greater the distance, the greater the X-direction shift.

This little piece of code introduces the outstanding MultiplyTransform method. This method appends a transformation Matrix object to the world transformation. In this case, the existing transformation is the identity matrix, so this method is a convenient way of allowing the Matrix object created to provide the world transformation.

WHY TRANSFORMATIONS ARE NOT APPLIED TO INVALIDATION AND CLIPPING OPERATIONS

Transformations are *not* applied to invalidation or clipping operations. These operations always happen in terms of device coordinates, not page coordinates or world coordinates. Another way to explain this dynamic is as follows:

- The `Graphics` object that is your drawing surface is created anew with each raising of the `Paint` event. You set transformations by calling methods on the `Graphics` object, and if you want to apply a particular transformation over multiple `Paint` events, you must retain that transformation as part of the state of your window or control, and reapply it each time your `Paint` event handler executes. Transformation state is not retained from `Paint` event to `Paint` event.
- You often need to invalidate rectangles and regions in an event handler other than the `Paint` event handler, such as a mouse event handler. In this case, the `Graphics` object that is your drawing surface hasn't even been instantiated yet.
- Invalidating or clipping to a partial pixel doesn't really make sense, and if you were invalidating or clipping with world coordinates, you would often be doing this.

The rectangles or region that you specify in a call to an `Invalidate` method are not transformed, and it is not possible to transform them. After invalidating, you will receive a `Paint` event, along with a `Graphics` object. You can apply transformations to the drawing surface, but the invalidated region does not change as you apply transformations.

 Previous

Next 

 Previous

Next 

Transformations and Custom Controls

In [Chapter 13](#), we'll revisit the subject of transformations in the context of building a scrolling control. However, it's fair to say that when building grid controls, text box controls, button controls, slider controls, bar charts, line graphs, and the like, transformations are not commonly used.

Having said that, here are a few areas in which you may reach for a transformation:

- If you are building a charting control that has the capability of panning in any direction, you could use the translation transformation so that you would not need to change the coordinates of the location where you draw the charting elements.
- If you are building a drawing custom control that gives users the capability to zoom in or out, you could use the scaling transformation to change the view of the custom control.
- In many cases, the print preview capability that comes with the .NET Framework is more than satisfactory for your purposes. However, if you have some additional requirements that necessitate building your own print preview dialog box, you could use transformations to implement the panning and zooming functionality.
- If you were building a custom clock control, or perhaps a custom temperature dial control, you might use the rotation transformation.

 Previous

Next 

 Previous

Next 

Summary

In this chapter, you saw how to apply global transformations, which convert from world coordinates to page coordinates. Behind the scenes, global transformations are applied via a matrix, but for the purposes of building custom controls, you don't need to be too concerned with the matrix. Rather, you can use some convenience methods to easily set up your transformations.

To summarize, when you do a drawing operation, the coordinates that you pass to the drawing operation

are floating point, expressed in the world coordinate system. GDI+ then applies any world transformations to the drawing operation. Then, if the page coordinate system is set to anything other than the Pixel unit of measure, GDI+ does the conversion to pixels. This is a scaling operation based on the page coordinate system unit of measure and the drawing surface resolution (DPI).

It is important to note that transformations don't apply to clipping invalidations. When you invalidate a rectangle (or a region), you specify coordinates in terms of pixels on the underlying drawing surface. They are expressed in device coordinates.

Specifically, in this chapter, you saw how to carry out transformations that allowed scaling, translating, and rotating. Additionally, we looked at how these operations can be used (carefully and in the appropriate order) to give cumulative effects. We also touched on the underlying matrix multiplication aspects of transformations and used this knowledge to aid in understanding reflection and shearing transformations.

Finally, I suggested some areas where transformations may be of use in custom control development.

In the [next chapter](#), we're going to move on to the subject of printing. You'll see the similarities and differences you encounter when sending a graphic to a printer instead of sending it to the screen.

 Previous

Next 

 PreviousNext 

Chapter 9: Printing

Overview

It's very common for applications to provide some facility to print some data or results to a printer; in many cases, this is a crucial part of the application. Imagine, for example, trying to market a word processor or a spreadsheet that had no printing capabilities. How many copies do you think you would sell?

So far in this book, the examples have been focused on drawing to the screen. As it turns out, the general principles of drawing to a printer are the same, to an extent. You still use a `Graphics` object to represent the drawing surface, and use its methods and properties to perform drawing operations. But of course, there are many differences between the screen and the printer.

When you're drawing custom controls for display on the screen, you are concerned about making sure that pixels are colored exactly as you want them to be. This is particularly important for controls with horizontal and vertical lines, since you don't want to create situations where a line overshoots or falls just short of a line perpendicular to it. When drawing to a printer, you're a little less concerned about this, because printers tend to have higher resolutions than screens. Therefore, this chapter will focus less on the details of how to draw individual items, and more on how to manipulate and write handlers for the various printing-related events.

There is a lot of printing-related functionality offered by Windows that doesn't have much to do with GDI+ (such as how the user selects a range of pages to print, setting the paper tray or bin, putting up status bars while printing, and so on). This chapter will focus primarily on the parts of the API that relate to using the printer as a drawing surface. This includes changing its dimensions, orientation, or resolution; and getting the printable area of the drawing surface. In addition, you'll see how to write a *single* set of code that draws what you want on *both* the screen and printed page, as well as what's involved in creating a print preview on the screen.

Our first example is probably the simplest printing program possible. It just displays a string and a rectangle, and allows the user to print these on a single page. We'll then extend the sample by having it spread its printout over two pages. Next, we'll examine graphics units and develop an application that uses graphics units to draw a ruler of the correct size on paper. Finally, we'll examine the metrics of the drawing surface for a printer and for the Print Preview dialog box. In the course of all this, you will also see examples of the programmatic manipulation of the rudimentary features of the Print dialog box, so that you know how to put up a dialog box that allows the user to select a printer, select pages to print, and so on.

This chapter covers the following topics:

- How to deal with a printer's capabilities, in terms of resolution and color, and the physical setup of the page
- How to allow your application's users to select various printer and page options, just as they can when printing a word-processed document or spreadsheet
- How to control multiple-page printing

We'll start off by recapping what users normally expect to happen when they elect to print a document.

 PreviousNext  PreviousNext 

The Printing Process

Before we examine how to take programmatic control of printing capabilities, it's worthwhile to recap the features that the user will normally expect to see. Typically, in an application that supports printing, there are three menu items that the user can select:

- **Page Setup:** When the user selects this option, the application presents a Page Setup dialog box to the user. This dialog box allows the user to select various options (such as margin size, whether to print in landscape or portrait mode, and so on). The module that puts up the Page Setup dialog box retains the results of the dialog box in a `PageSettings` object.
- **Print:** When the user selects this option, the application presents a printer selection dialog box. The user selects the desired printer and clicks OK. The application then starts the printing process, using the page settings.
- **Print Preview:** When the user selects this option, the application opens the Print Preview dialog box, using whatever default or custom page settings have been applied.

If your application will support printing, you will normally want it to conform to user expectations, so you will need to provide these three menu options. Fortunately, Microsoft has made this task easy by providing a number of .NET Framework classes that implement much of the boilerplate code associated with these tasks. These classes are able to store printer settings, display the standard Page Setup and Print dialog boxes, and show the print preview. Normally, using these classes, the only functionality that you must implement yourself is the code that tells the computer exactly what to print.

The Printing-Related Classes in GDI+

Most of the GDI+ printing-related classes are in the `System.Drawing.Printing` namespace and are implemented in the `System.Drawing.dll` assembly. However, some classes that represent dialog boxes or forms are in the `System.Windows.Forms` namespace and the `System.Windows.Forms.dll` assembly.

There are a number of classes that you will use when writing code to print. Here, we will briefly review the six main classes in GDI+ that are of particular interest: three of them are dialog boxes, and the other three control the printing process itself. [Table 9-1](#) lists these classes, along with brief descriptions.

Table 9-1: Printing-Related Classes

Namespace	Class	Description
<code>System.Windows.Forms</code>	<code>PageSetupDialog</code>	Represents the Page Setup dialog box. Contains properties that give you programmatic control of margins and page orientation, various page settings, and printer settings (see the <code>PageSettings</code> and <code>PrinterSettings</code> classes).
<code>System.Windows.Forms</code>	<code>PrintDialog</code>	Represents the Print dialog box, which is displayed immediately before printing. Contains properties that give you programmatic control of printer settings (see the <code>PrinterSettings</code> class) and various other printing capabilities.
<code>System.Windows.Forms</code>	<code>PrintPreviewDialog</code>	Represents the Print Preview dialog box. Gives you programmatic control of print preview functionality, such as zooming in and out, scrolling, etc.
<code>System.Drawing.Printing</code>	<code>PageSettings</code>	Encapsulates characteristics that relate to a single printed page, and which may change from page to page. This is a child of the <code>PageSetupDialog</code> class, because we tend to use that dialog box to control page properties.
<code>System.Drawing.Printing</code>	<code>PrinterSettings</code>	Encapsulates some characteristics of a specified printer, such as whether the printer is a color printer, whether it

		supports landscape mode, and so on. This is a child of the <code>PageSetupDialog</code> and <code>PrintDialog</code> classes, because we tend to use those dialog boxes to control page properties. Unless you are doing a special implementation of printing (such as forcing output to a particular printer), you don't need to use this class.
System.Drawing.Printing	PrintDocument	Encapsulates a print document in memory. You use this object to send output to a Print or Print Preview dialog box.

How a Document is Printed or Print-Previewed

In this section, we'll review what actually happens—as far as a program is concerned—when the user opts to print or print preview. We'll cover printing first.

What Happens When a User Elects to Print

The printing process itself is normally controlled with a `PrintDocument` class. In your code, you can invoke the printing process (that is, send the document to the printer for printing) by calling the `PrintDocument.Print` method. This is what you would do after the user clicked OK in the Print dialog box. The `Print` method controls the printing by raising a number of events. Your application will need event handlers for some or all of these events. The sequence of events is as follows:

1. The `BeginPrint` event is raised. Your application's `BeginPrint` event handler should perform any special processing required when printing starts.
2. The `QueryPageSettings` event is raised. The event handler for this event should make any changes required to printer settings that relate to *just* that page.
3. The `PrintPage` event is called. This is arguably the most important of all of these events. The handler for this event should do the actual printing of a page. The `PrintPage` event is similar to the `Paint` event, but with two important differences. First, while the `Paint` event handler is passed an argument of type `PaintEventArgs`, the `PrintPage` event handler expects an argument of type `PrintPageEventArgs`. This `PrintPageEventArgs` object contains extra information about the page to be printed. Second, the `Graphics` object associated with the `PrintPageEventArgs` object will have been initialized to point to the printer. Thus, the `Graphics` object represents a *printer* drawing surface, instead of a screen drawing surface. In addition to printing, the `PrintPage` event handler should also set the `PrintPageEventArgs.HasMorePages` property to `true` or `false` depending on whether any pages remain to be printed.
4. If `PrintPageEventArgs.HasMorePages` is set to `true` (on return from the `PrintPage` event handler), the `PrintDocument` class will go back to raising the `QueryPageSettings` event (step 2), followed by the `PrintPage` event (step 3). This loop will continue until `HasMorePages` is set to `false`. Clearly, this means that your code will need to independently keep track of how many times `PrintPage` has been called in order to determine which page is being printed, and to customize your printing code appropriately.
5. Finally, the `EndPrint` event is called to perform any cleanup needed when printing has finished.

In this chapter, we will mostly use the `PrintPage` event. Later in the chapter, we will work through an example that involves an event handler for `QueryPageSettings`, but we won't use either `BeginPrint` or `EndPrint`. Nevertheless, it's useful to know that these events are there if you need them.

What Happens When the User Elects to Print Preview

The process for print previewing is very similar to the process for printing, but there are one or two differences.

The `PrintPreviewDialog` has a `Document` property, which you set to refer to the `PrintDocument` object that controls printing in your application.

In the case of print previewing, the process is invoked by a call to the `PrintPreviewDialog` object's `ShowDialog` method, which results in the Print Preview dialog box being displayed. In contrast, in the case of printing, you use the `PrintDialog` object's `ShowDialog` method to show the Print dialog box, and then the printing process *itself* begins when the user clicks OK in this dialog box to invoke the `PrintDocument.Print` method.

Internally, the `PrintPreviewDialog.ShowDialog` method will cause the same sequence of events to be raised against the contained `PrintDocument` object. However, it will ensure that the `Graphics` object is initialized to send output to the *print preview* window, rather than to the printer. This means that you can normally reuse exactly the *same* drawing code for both printing and print previewing a document.

How the Printing Classes are Used

Now that we've reviewed the printing process from a programmatic perspective, let's see how the main printing classes are normally used.

In the class that implements the event handlers for the menu items (often the main `Form` class), you declare a `PageSettings` member variable. This variable will store the current page settings for the application.

The event handler for the Page Setup menu item first creates a `PageSetupDialog` object to represent the dialog box. You assign the object's `PageSettings` property to be your `PageSettings` member variable, and use the stored page settings in the dialog box. You show the dialog box by calling its `ShowDialog` method.

The event handler for the Print menu item first creates a `PrintDocument` object to represent the dialog box. Then it registers one or more event handlers with the object (at the least, an event handler must be registered for the `PrintPage` event). You assign the `PrintDocument` object's `DefaultPageSettings` property to be your `PageSettings` member variable, so that (to begin) the dialog box uses the page settings stored in state. The `PrintPage` event handler then puts up a Print dialog box using the `PrintDialog.ShowDialog` method, and if the user clicks OK in the dialog box, starts the printing process. In this chapter, we'll explore the `PrintDialog` class in detail.

In a similar fashion, the event handler for the Print Preview menu item creates an instance of the `PrintDocument` class, and registers one or more event handlers with the object (again, at the least, it must register an event handler for the `PrintPage` event). Again, you assign the `PrintDocument` object's `DefaultPageSettings` property to be your `PageSettings` member variable, so that (to begin) the dialog box uses the page settings stored in state. The `PrintPage` event handler then puts up a Print Preview dialog box using the `PrintPreviewDialog.ShowDialog` method.

As we've discussed, the `PrintPage` event handler will get a `Graphics` object as an argument to the method, and it can draw on the drawing surface using this object. The .NET Framework will raise this event when it is ready for the event to draw on the printed page. Before returning, this event handler can determine if there are more pages to be printed, and indicate this to its caller. The application must use its own algorithm and knowledge of what it is printing to determine if more needs to be printed. If the application indicates that there are more pages, the `PrintPage` event will be raised again. If it indicates that there are no more pages, the printing process terminates, and `PrintPage` will not be raised again.

The print preview process uses the same events as the print process. If you write your application so that your code behaves properly, regardless of the resolution of the drawing surface, your printing code will also serve for the print preview functionality.

 Previous

Next 

 Previous

Next 

A Simple Printing Example

Now that you've met the six classes in GDI+ that provide the printing functionality set, we'll build a little

application to see them in action. Our application, called `SimplePrintingExample`, will simply display some text and a box, as shown in [Figure 9-1](#). However, what is new is that this application also enables the user to print the contents of the form to a printer, and view it in a Print Preview dialog box.

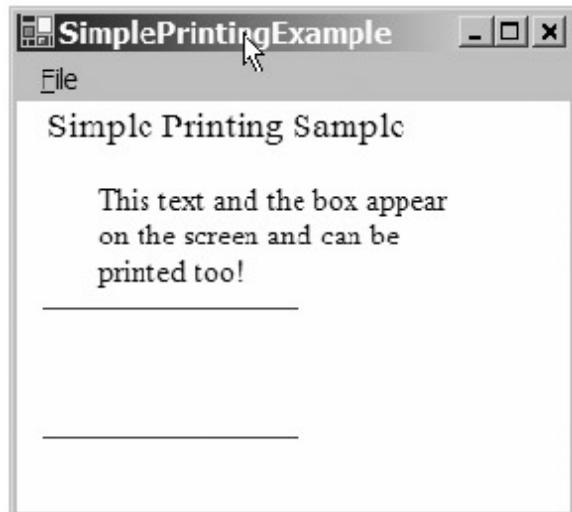


Figure 9-1: Simple printing application

Setting up the Windows Application

To start, we create a C# Windows Application in Visual Studio .NET. In the `InitializeComponent` routine (or via the Design view if you prefer), set the background color of the form to the color `SystemColors.Window`. On most machines, this will appear as white (though it may be different if you have set up a nondefault color scheme). You can also set the `Window` text here if you like:

```
private void InitializeComponent()
{
    ...
    this.BackColor = System.Drawing.SystemColors.Window;
    this.Text = "SimplePrintingExample";
}
```

Now add an event handler for the form's `Paint` event, and add the following code to it:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    PaintDocument(g);
}
```

The painting itself is done in a new helper method called `PaintDocument`, which we'll define next. The reason for factoring this code out into a different method is so that we can call it both when printing *and* when painting to the screen:

```
private void PaintDocument(Graphics g)
{
    g.GraphicsUnit = GraphicsUnit.Point;

    // Draw some items to the drawing surface
    g.DrawString("Simple Printing Sample",
        this.mainTextFont,
        Brushes.Black,
        new Rectangle(10, 20, 180, 30));
    g.DrawString("This text and the box appear on the screen " +
        "and can be printed too!",
        this.subTextFont,
        Brushes.Black,
        new Rectangle(30, 50, 150, 50));
```

```
g.DrawRectangle(Pens.Blue,
    new Rectangle(new Point(10, 100), new Size(100, 50)));
}
```

As you can see, this code simply draws the two text strings and the rectangle. It requires two `Font` member variables of the `Form` class, which we'll also define:

```
public partial class Form1 : Form
{
    private Font mainTextFont = new Font("Times New Roman", 14);
    private Font subTextFont = new Font("Times New Roman", 12);
```

The only other point worth noting is that we have changed the page unit in our `PaintDocument` routine above:

```
g.GraphicsUnit = GraphicsUnit.Point;
```

Setting this property to `GraphicsUnit.Point` means that measurements will be made in printing points (1/72 inch). The default is to make all measurements in pixels, but if we do that, the rectangle will almost certainly appear too small when we print it, because a pixel on a printer occupies a far smaller area than a pixel on the screen. By setting the page unit to `Point`, we will ensure that the unit of measurement remains similar when we swap from screen to printer. Don't worry about this too much for now; we'll explore page units in more detail in the "[Units of Measurement](#)" section later in this chapter.

With these code changes, the `SimplePrintingExample` program will run and display on the screen correctly. However, it doesn't yet have any printing functionality, so we need to add that next.

Adding Printing Event and Menu Handlers

First, we need to reference the appropriate namespace in the code.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Printing;
```

Next, use the Visual Studio .NET Design view editor to add a File menu with Page Setup, Print Preview, and Print menu options, as shown in [Figure 9-2](#).

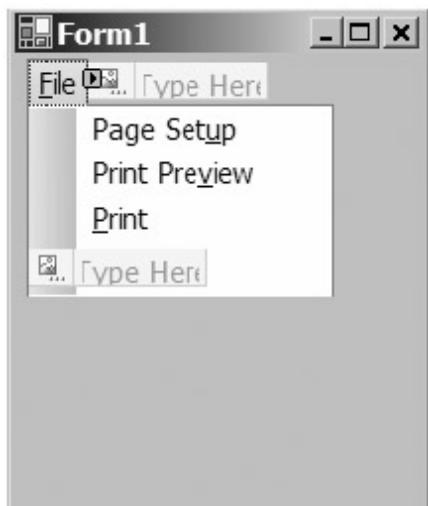


Figure 9-2: Setting up the menu

Use the Properties window for these items to give them the names `menuFilePageSetup`, `menuFilePrintPreview`, and `menuFilePrint`.

Also add event handlers for these menu items. We'll give these handlers the respective names

menuFilePageSetup_Click, menuFilePrintPreview_Click, and menuFilePrint_Click. Now let's build the code for these three event handlers.

The Page Setup Menu Event Handler

Let's start by looking at the Page Setup event handler. Recall that this needs to display a standard Page Setup dialog box and save any preferences the user chooses. First, we need a new member variable (a `PageSettings` object) in the `Form1` class to hold the saved settings.

```
public partial class Form1 : Form
{
    private Font mainTextFont = new Font("Times New Roman", 14);
    private Font subTextFont = new Font("Times New Roman", 12);
    private PageSettings storedPageSettings;
```

Now, let's move on to the code for the event handler.

```
private void menuFilePageSetup_Click(object sender, System.EventArgs e)
{
    try
    {

        PageSetupDialog psDlg = new PageSetupDialog();

        // Create a PageSettings object if never before created
        if (this.storedPageSettings == null )
            this.storedPageSettings = new PageSettings();

        // Put up the dialog
        psDlg.PageSettings = this.storedPageSettings;
        psDlg.ShowDialog();

    }
    catch(Exception ex)
    {
        MessageBox.Show( ex.Message );
    }
}
```

This code should be fairly self-explanatory. We instantiate a `PageSetupDialog` object and set its `PageSettings` property to refer to the stored settings. Then we call its `ShowDialog` method to put the dialog box on the screen. This is literally all we need to do.

At this stage, you can run the program and check that this works. Select the Page Setup option in the menu, and you should see the Page Setup dialog box, as shown in [Figure 9-3](#), which shows the default settings. (The sample page shown in the dialog box is a fixed schematic supplied by Microsoft; it's not drawn using any of your code.)



Figure 9-3: Page Setup dialog box

The Print Event Handler

Next, we'll implement the Print menu handler.

```
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    try
    {
        PrintDocument pd = new PrintDocument();
        pd.PrintPage += 
            new PrintPageEventHandler( this.PrintPageEventHandler );

        // If the user has set page settings, then set the property
        // in the print document
        if ( this.storedPageSettings != null )
            pd.DefaultPageSettings = this.storedPageSettings;
        PrintDialog dlg = new PrintDialog();
        dlg.Document = pd;
        DialogResult result = dlg.ShowDialog();
        if (result == System.Windows.Forms.DialogResult.OK)
            pd.Print();
    }
    catch ( Exception ex )
    {
        MessageBox.Show( ex.Message );
    }
}
```

In this event handler, first we instantiate a `PrintDocument` object that will control the printing. We attach the handler responsible for printing each page to the `PrintDocument`'s `PrintPage` event. In this case, the handler is also called `PrintPageEventHandler`. (We haven't built that yet, but we'll do so in minute.)

Next, we set the `PrintDocument`'s `DefaultPageSettings` property to refer to our `storedPageSettings` member object, which holds any settings the user has stored. Then we instantiate a `PrintDialog` object, set it to refer to the newly created `PrintDocument`, and call its `ShowDialog` method. That will result in the normal Print dialog box being shown.

When the user clicks OK in the Print dialog box, the document will be printed. During the printing, the system will repeatedly raise the `PrintPage` event, so we need to code the corresponding event handler (our `PrintPageEventHandler` routine). Here's what the code looks like:

```

protected void PrintPageEventHandler( Object obj, PrintPageEventArgs ev )
{
    Graphics g = ev.Graphics;
    PaintDocument(g);
    ev.HasMorePages = false;
}

```

Note that this event handler takes `PrintPageEventArgs` as a parameter. As explained earlier, this class is similar to `PaintEventArgs`, but it contains additional information related to printing. Of most immediate interest to us is the Boolean property `HasMorePages`, which is used to determine whether the `PrintPage` event will be raised again. For this particular document, there is only one page to be printed, so we set this property to `false`.

Other than that, this event handler looks pretty much like our original `Paint` event handler: it calls the `PaintDocument` method to display the text and rectangle. The real magic of this is that the system will ensure that the `Graphics` object supplied to the event handler has been set up to direct all painting to the *printer* instead of the screen. So, we can just call the `PaintDocument` method, without worrying about where the result of the `Paint` operation will go.

Note In this simple example, we've used the same routine, `PaintDocument`, to print *and* to paint.

We have been able to do this because our application is so simple. It has only one page, and we want the printed document to look the same as the form on the screen. However, in practice, there will be differences between printing and painting. In particular, for longer documents, when painting to the screen, you use scroll bars to just print the appropriate portion, while when printing, you simply print one page at a time. These differences may mean that you need to write separate methods to paint to the screen and to print.

The Print Dialog Box

[Figure 9-4](#) shows the Print dialog box that our application displays. It should look very familiar.

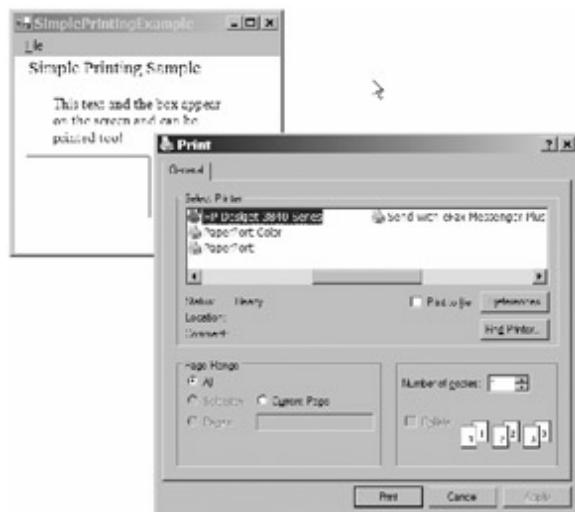


Figure 9-4: The Print dialog box

Note that, by default, the option for the user to print only a selection from the document is disabled. For our example, it makes sense to leave it that way, because there is only one page anyway, and no facility for the user to select a region on the screen. If you wish, you can enable these controls by respectively setting the `PrintDialog.AllowSomePages` and `PrintDialog.AllowSelection` properties to `true` before putting up the dialog box.

The Print Preview Event Handler

Now, let's look at the `PrintPreview` event handler. This event handler is very similar to the `Print` event handler. The two differences are highlighted in the following code.

```

private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{

```

```
try
{
    PrintDocument pd = new PrintDocument();
    pd.PrintPage +=
        new PrintPageEventHandler( this.PrintPageEventHandler );

    // If the user has set page settings, then set the property
    // in the print document
    if ( this.storedPageSettings != null )
        pd.DefaultPageSettings = this.storedPageSettings;
    PrintPreviewDialog dlg = new PrintPreviewDialog();
    dlg.Document = pd;
    dlg.ShowDialog();
    // No separate command to print.
    // Preview dialog shows the print preview
}
catch ( Exception ex )
{
    MessageBox.Show( ex.Message );
}
}
```

We instantiate a `PrintPreviewDialog` object, instead of a `PrintDialog` object. Also, we send the document straight to the Print Preview dialog box via its `ShowDialog` method, instead of waiting for the result of a Print dialog box, and then sending the document to the printer.

The Print Preview Dialog Box

When we open the Print Preview dialog box, we see a preview of our document, as shown in [Figure 9-5](#). The `PrintPreviewDialog` class that has been supplied by Microsoft is very sophisticated. We don't need to do any programming ourselves; it provides for the user to be able to take several actions, including the following:

- Print directly to the printer from the Print Preview dialog box.
- Zoom in or out on our document.
- Pan around the document, if zoomed in close enough.
- Select the number of pages that are shown simultaneously (1, 2, 3, 4, or 6). There are five buttons at the top of the dialog box for this, as you can see in [Figure 9-5](#). (If you select to display more pages than are actually in the document, the dialog box will just display the pages that are available.)
- Go directly to a specific page.

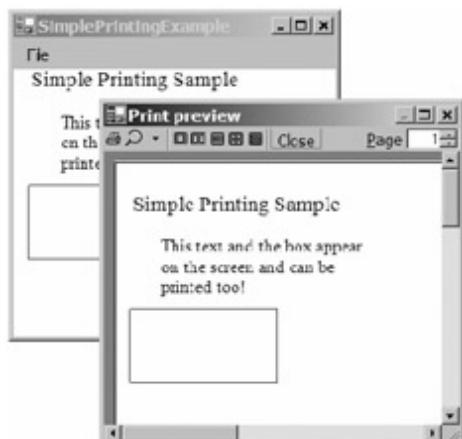


Figure 9-5: Print preview dialog box

Printing to More than One Page

The SimplePrintingExample program demonstrates how events related to printing are raised, and how you can print and print preview by supplying the event handler for the PrintPage event. However, there is one respect in which it is too simplistic: it prints onto only one page. In the real world, it is far more likely that your application's printout will take up several pages. Therefore, in this section, we will develop SimplePrintingExample a bit further, so that its output is spread over two pages, thus illustrating how you can control *multipage printing*. We'll rename it MultiPagePrintingExample and make some changes.

The currentPage Variable

First, we add a member variable to the Form1 class. This variable will store the count of which page we are up to in any printing job.

```
public partial class Form1 : Form
{
    private Font mainTextFont = new Font("Times New Roman", 14);
    private Font subTextFont = new Font("Times New Roman", 12);
    private PageSettings storedPageSettings;
    uint currentPage;
```

The Revised Print and Print Preview Event Handlers

In the event handler for the File ➤ Print menu option click event, we initialize this variable to indicate Page 1:

```
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    try
    {
        this.currentPage = 1;
        PrintDocument pd = new PrintDocument();
        pd.PrintPage +=
            new PrintPageEventHandler( this.PrintPageEventHandler );
        // If the user has set page settings, then set the property
        // in the print document
        if ( this.storedPageSettings != null )
            pd.DefaultPageSettings = this.storedPageSettings;
        PrintDialog dlg = new PrintDialog();
        dlg.AllowSomePages = true;
        dlg.Document = pd;
        DialogResult result = dlg.ShowDialog();
        if (result == System.Windows.Forms.DialogResult.OK)
            pd.Print();
    }
    catch ( Exception ex )
    {
        MessageBox.Show( ex.Message );
    }
}
```

Note that we've also set the AllowSomePages property of the PrintDialog object to true, to enable the user to choose to print only some pages.

We also need to initialize the page in the print preview event handler, as follows:

```
private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{
    try
    {
        this.currentPage = 1;
        PrintDocument pd = new PrintDocument();
```

The Revised PrintPage Event Handler

What is being printed is no longer identical to what appears on the screen: the screen shows the text and

rectangle together, but when printing, these appear on separate pages. Therefore, we are no longer able to combine the painting code for the two scenarios. Instead, we will need to write separate drawing code in the `PrintPage` event handler that detects the current page and responds appropriately. Here's how we do it:

```
protected void PrintPageEventHandler( Object obj, PrintPageEventArgs ev )
{
    Graphics g = ev.Graphics;
    g.GraphicsUnit = GraphicsUnit.Point;

    switch (currentPage)
    {
        case 1:
            g.DrawString("Simple Printing Example",
                        this.mainTextFont,
                        Brushes.Black,
                        new Rectangle(10, 10, 180, 30));
            g.DrawString("This text and the box appear on the screen " +
                        "and can be printed too!",
                        this.subTextFont,
                        Brushes.Black,
                        new Rectangle(30, 40, 150, 50));
            ev.HasMorePages = true;
            ++currentPage;
            break;
        case 2:
            g.DrawRectangle(Pens.Blue,
                           new Rectangle(new Point(10, 90), new Size(100, 50)));
            ev.HasMorePages = false;
            break;
        default:
            Debug.Assert(false);
            break;
    }
}
```

Note that for the `Debug.Assert` method, you'll need to add a reference to the `System.Diagnostics` namespace:

```
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Diagnostics;
```

To appreciate what we've done here, it's worth comparing this code to the `PaintDocument` and `PrintPageEventHandler` methods of the `SimplePrintingExample` application. This routine contains a little code from both of those methods, plus some new code.

We retrieve the `Graphics` object that will be used to print or print preview, and set the page unit to `Point`. What we do next depends on the page being printed:

- If `currentPage` is 1, we draw the text and then increment `currentPage` so it will have the correct value the next time `PrintPageEventHandler` is called.
- If `currentPage` is 2, we draw the rectangle and set the `HasMorePages` property of `PrintPageEventArgs` to `false`, to make sure that the `PrintPage` event is not raised again.

Note that the `PrintPageEventArgs` object itself doesn't store any record of the page that is to be printed. (It would be nice if it did, because it would have saved us from needing to maintain this record separately in our code.) All the `PrintPageEventArgs` object can do is indicate whether more pages remain to be printed.

Also note that this `PrintPageEventHandler` routine is still rather inflexible, in that we have determined there will be exactly two pages. In a more sophisticated application, you will need to write code that examines the data being printed at runtime, to determine how many pages should be printed and what should be printed on what page. This will depend on how much data can be printed on a page, and this, in turn, depends on the printer settings (page size, margins, and so on). You will see how to access this

data in the "[Printing Metrics](#)" section later in the chapter. By modifying the example a bit, you can make your application print any number of pages.

Figure 9-6 shows an example of the Print Preview dialog box displayed by the `MultiPagePrintingExample` application. In this figure, I have opted to display four pages, but the program has correctly detected (by repeatedly raising the `PrintPage` event) that there are only two pages to be printed.

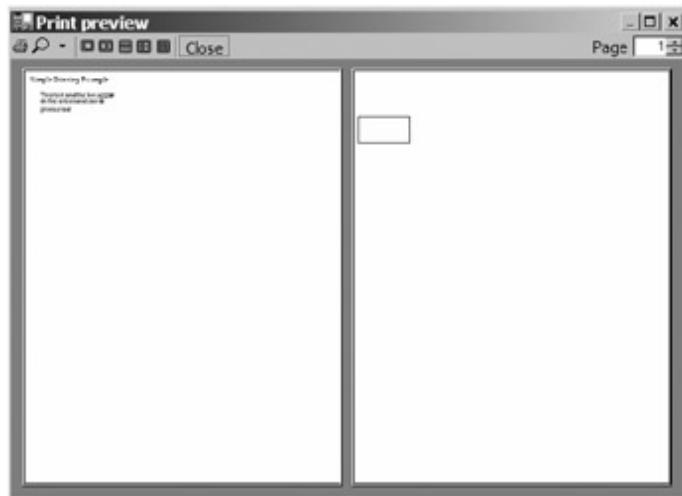


Figure 9-6: Print preview of two pages

Previous

Next

Previous

Next

Units of Measurement

In this section, we will take a closer look at measurement units. This actually is part of the larger topic of page transformations and the world coordinate system, which was introduced in [Chapter 8](#). Here, we'll examine the measurement units in relation to printing.

Painting vs. Printing Measurements

By default, when painting in response to a `Paint` event, the graphics unit is set to ensure measurements are in pixels for drawing shapes and images (though not for text fonts, as you'll see in this section). So, for example, the following code will display a square that is 100 pixels by 100 pixels:

```
private void PaintDocument(Graphics g)
{
    ...
    g.GraphicsUnit = GraphicsUnit.Pixel;           // can omit this line
                                         // line as Pixel is the default anyway
    Rectangle cr = new Rectangle(new Point(10,90), new Size(100, 100));
    g.DrawRectangle(Pens.Blue, cr);
}
```

Using pixels as a measurement is often extremely convenient for displaying to the screen, because a pixel is the smallest unit that can be drawn on the screen. By specifying a line width for a horizontal or vertical line as an integer number of pixels, for example, you know that you are specifying a line width that can be drawn exactly, resulting in a clear image on the screen that can be drawn without aliasing.

Unfortunately, when it comes to *printing*, measuring sizes in pixels causes problems, because a pixel is considerably smaller on a printer than it is on the screen. On the screen, the preceding code will result in a square with sides of approximately 1 inch, depending on the exact resolution of the monitor. But many printers come with resolutions of over 1000 DPI (dots per inch, the number of pixels per inch). If you use the preceding code to print a square on the printer, the square will still have sides of 100 pixels each, but on a printer with a resolution of 1200 DPI, this will be less than 0.1 inch! So, if you use pixels for your

measurements, it becomes a lot harder to write applications whose printouts look the same as the graphic displayed on the screen.

Fortunately, we can get around this by setting the `Graphics.PageUnit` property.

Setting the PageUnit Property of the Graphics Object

The `Graphics.PageUnit` property allows you to use measurements in some unit that remains roughly the same on the screen and printer. You can set it to any of the values listed in [Table 9-2](#) from the `GraphicsUnit` enumeration (which is part of the `System.Drawing` namespace).

Table 9-2: Graphics.PageUnit Values

Value	Meaning
Pixel	Smallest unit into which an image can be broken down; typically, 1/96 inch on a display device, and anything from 1/300 to 1/1000 or less on a printer
Inch	An inch
Display	1/96 inch
Point	A standard point as traditionally used by printers, equal to 1/72 inch
Document	1/300 inch
World	A unit determined by the <code>Graphics</code> object's world transformation

The following code will draw a square with sides roughly 1 inch, regardless of whether it is drawn to the screen or the printer:

```
private void PaintDocument(Graphics g)
{
    ...
    g.PageUnit = GraphicsUnit.Inch;
    RectangleF cr = new RectangleF(new Point(0.4f, 0.9f),
                                   new Size(1.0f, 1.0f));
    g.DrawRectangle(Pens.Blue, cr);
}
```

The rectangle will be located roughly 0.4 inch horizontally and 0.9 inch vertically from the top left corner. Note that because we are using inches, we need to express measurements in fractional numbers, and hence need to use a `RectangleF` object rather than a `Rectangle` object.

An unfortunate anomaly is that screen measurements are not always precisely correct. By default, all screen resolutions are defined to be 96 DPI (you can check this in the Settings tab of your windows desktop Properties dialog box, by clicking the Advanced button). However, different monitors have different sizes, and so they don't always display 96 pixels to the inch. Therefore, that measurement is only approximate. On the other hand, the computer is able to query a printer for its precise DPI, so measurements do normally come out exactly as specified on the printed page.

There are three separate areas in which you need to pay particular attention to the `PageUnit` property of the `Graphics` object: when you're drawing lines and filling shapes, drawing text, and drawing images.

Units for Drawing Lines and Filling Shapes

Coordinates that you pass to drawing methods are in the units of the `PageUnit` property. As mentioned, the default setting for `PageUnit` is `GraphicsUnit.Display`. By default, all coordinates are expressed in 1/96 inch.

One point that may not be immediately obvious is that pen widths are also expressed in the same units. If you use a pen width of 1, this means that the pen width will be 1/96 inch. If your printer has a resolution of 300 DPI, the pen will be about 3 pixels wide.

Units for Drawing Text

With `Font` objects, the `Font.Unit` property is the `GraphicsUnit` for text drawn with the font. This means that you can use one scale for drawing lines and filling shapes, and another scale for drawing text.

Thus, you could conceivably have a method that draws some text at a position expressed in the units stored in `Graphics.GraphicsUnit`, although the actual size of the text drawn is based on the units stored in `Font.Unit`.

The default value for `Font.Unit` is `GraphicsUnit.Point`, which is defined to be 1/72 inch. If you let the `Graphics.GraphicsUnit` property remain set to `GraphicsUnit.Display`, and you let the `Font.Unit` property remain set to `GraphicsUnit.Point`, you can accurately predict how your drawing operations will affect the drawing surface.

Alternatively, you could set the `Graphics.GraphicsUnit` property to `GraphicsUnit.Inch`, and also set the `Font.Unit` property to `GraphicsUnit.Inch`. In this case, all operations use the same unit of measure, which may simplify your drawing code. If you use this technique, there is one point to remember: pen widths will also be expressed in terms of inches. If you create a pen with a width of 1, this means that your pen will be 1 inch wide! However, this is easy to rectify, since you can express pen widths as floating-point numbers. If you set the pen width to 1/96, you will get the same results as if you had left `Graphics.GraphicsUnit` set to `GraphicsUnit.Display`.

Units for Drawing Images

When drawing an image, you specify the destination rectangle in terms of `Graphics.GraphicsUnit`. This means that GDI+ will almost certainly need to scale the image.

Drawing Using Inches

In this section, we will develop a sample application, called `RulerPrintingExample`, which illustrates use of the `Inch` as the graphics measurement unit. Here's what we need to do when using this technique.

- Set the `Graphics.GraphicsUnit` property to `GraphicsUnit.Inch`. We do this both when printing and when drawing to the screen.
- Set the pen width to 1/96 inch. As noted earlier, when printing, this has a nearly identical effect as using a pen width of 1 when the `Graphics.GraphicsUnit` property is set to `GraphicsUnit.Device`. However, because the screen is defined as having a resolution of 96 DPI, this results in creating a pen with a width of exactly 1 pixel. This removes any chance of aliasing; or if anti-aliasing is turned on, this removes the possibility of antialiasing artifacts.

This example actually draws a ruler, which will appear roughly the same when drawn in a window or printed to a page. I say "roughly" because, as previously noted, depending on your monitor, the measurements may not be exactly correct on the screen, although they will be on the printed copy.

The code for this example is almost identical to that for the earlier `SimplePrintingExample`. In fact, the only difference is in the implementation of the `PaintDocument` method:

```
private void PaintDocument( Graphics g )
{
    g.GraphicsUnit = GraphicsUnit.Inch;
    g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;

    float offset = .75f;
    Pen p = new Pen(Color.Black, 1.0f / 96.0f);
    for (float rule = 0; rule <= 5.0f; rule += .25f)
    {
        float x = offset + rule;
        float len;
        if (rule == (int)rule)
        {
            String s = ((int)rule).ToString();
            len = 0.5f;
            System.Drawing.Font f =
                new Font( "Times New Roman", 0.25f, GraphicsUnit.Inch );
            StringFormat sf = new StringFormat();
            sf.Alignment = StringAlignment.Center;
            RectangleF mr = new RectangleF(x - .25f, 1.5f, .5f, f.Height);
            g.DrawString(s, f, Brushes.Black, mr);
        }
    }
}
```

```
        g.DrawString(s, f, Brushes.Black, mr, sf);
    }
    else if (rule * 2 == (int)(rule * 2))
        len = 0.375f;
    else
        len = 0.25f;
    g.DrawLine(p, x, .75f, x, .75f + len);
}
}
```

Running the `RulerPrintingExample` application results in the ruler appearing in the window, as shown in [Figure 9-7](#).

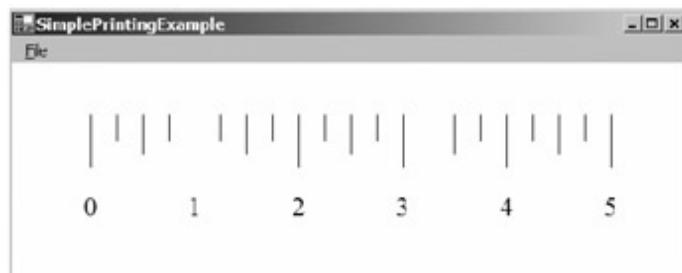


Figure 9-7: A printable ruler on the screen

The print preview results look like [Figure 9-8](#).

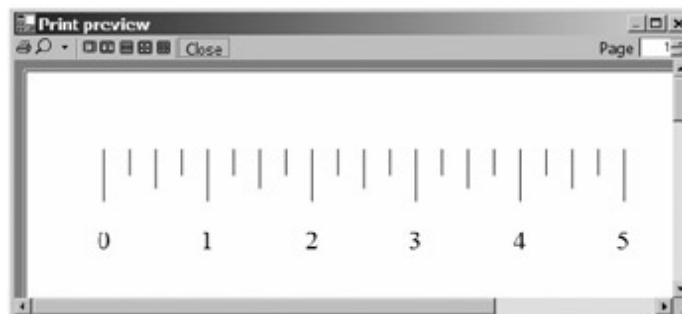


Figure 9-8: Print preview of the ruler

Previous

Next

Previous

Next

Printing Metrics

We use the term *printing metrics* to describe the data that determines how the printer actually prints a document. It includes such measurements as the *printer resolution* (that DPI value we've talked about), the *paper size*, and the *page orientation*.

It is sometimes important to be able to read certain metrics within the code that prepares your document for printing or print previewing. You can use these metrics to determine how you will print the contents of the page. For example, if you are writing code that is designed to print large spreadsheets or documents, you will need to use the paper size to calculate exactly how much text you can fit on a page, and hence where the page breaks should occur and how many pages you need to print.

Some metrics can be changed in the Page Setup dialog box. These are obtained from the `PageSettings` object. Other metrics are passed to the `PrintPage` event handler, and these are obtained from the `PrintPageEventArgs` object. There are other metrics that are obtained from the supplied `Graphics` object itself.

We'll consider these metrics in two groups: those that relate to the *printer*, and those that relate to the *printed page*. [Table 9-3](#) shows the printer metrics, and [Table 9-4](#) shows the printed page metrics.

Table 9-3: Printer Metrics

Metric	Description
PageSettings.PaperSize	This property returns a <code>PaperSize</code> object, whose properties (<code>Height</code> , <code>Width</code> , <code>Kind</code> , and <code>PaperName</code>) tell us about the type of paper that the printer says it will print to. The dimensions (<code>Height</code> and <code>Width</code>) are in units of 1/100 inch. The <code>Height</code> property refers to the long side of the paper (therefore, these dimensions are not affected by changing the page orientation between <code>Portrait</code> and <code>Landscape</code>).
PageSettings.PrinterResolution	This property returns a <code>PrinterResolution</code> object, whose properties (<code>Kind</code> , <code>X</code> , and <code>Y</code>) describe the resolution of the printer. The <code>Kind</code> property returns a value from the <code>PrinterResolutionKind</code> enumeration (<code>High</code> , <code>Medium</code> , <code>Low</code> , <code>Custom</code> , or <code>Draft</code>), and the <code>X</code> and <code>Y</code> properties are integers.
PageSettings.PrinterSettings.LandscapeAngle	If the printer supports landscape printing, this value represents the angle of rotation that the portrait orientation is rotated in order to get the landscape orientation. In this case, the value is either 90 or 270 (degrees). (Note that the printer will print portrait mode unless the <code>PageSettings.Landscape</code> property is set to <code>true</code> .) If the printer does not support landscape printing, then this value is 0.

Table 9-4: Printer Page Metrics

Metric	Description
PageSettings.Bounds	This is a <code>Rectangle</code> object that reflects the dimensions of the drawing surface. These dimensions are always specified in 1/100-inch units, independently of the <code>Graphics</code> object's <code>PageUnit</code> property. These dimensions are affected by a change in the page orientation.
PrintPageEventArgs.PageBounds	This metric is the same as <code>PageSettings.Bounds</code> , and is supplied for convenience only.
PageSettings.Margins	This property returns a <code>Margins</code> object, whose properties (<code>Top</code> , <code>Bottom</code> , <code>Left</code> , and <code>Right</code>) report the margins that the user has specified in the Page Setup dialog box (in 1/100-inch units). If possible, you should honor this setting.
PrintPageEventArgs.MarginBounds	This is a <code>Rectangle</code> object that represents the area of the page within the margins. It is precomputed for convenience by taking the value of <code>PageSettings.Bounds</code> and decrementing the dimensions by the values in

	<code>PageSettings.Margins.</code>
<code>Graphics.DpiX</code>	This reports the physical horizontal resolution of the drawing surface.
<code>Graphics.DpiY</code>	This reports the physical vertical resolution of the drawing surface.
<code>Graphics.VisibleClipBounds</code>	This is a <code>RectangleF</code> object that reports the drawable area of the drawing surface. Some printers don't allow you to draw right up to the edge of the drawing surface, due to physical constraints of the printer. In some cases, if the clipping region has not previously been set, this property does not contain a valid value. See the following section for a workaround.

Displaying Printing Metrics

We will now write an example, called `PrintingMetricsExample`, which writes out the main printing metrics to the console window. The easiest place from which to access the printing metrics is within the `PrintPage` event handler. This event handler has access to the `PrintPageEventArgs` object, which in turn holds all the printer settings, either directly or via the `Graphics` or `PageSettings` objects, which are supplied as properties of the `PrintPageEventArgs` object.

The code in this example is identical to that in the `SimplePrintingExample` application, except in two places. First, we modify the `PrintPageEventHandler`, as follows:

```
protected void PrintPageEventHandler( Object obj, PrintPageEventArgs ev )
{
    WriteMetricsToConsole(ev);
    Graphics g = ev.Graphics;
    PaintDocument(g);
    ev.HasMorePages = false;
}
```

Second, we add the code for the `WriteMetricsToConsole` method that is called here. This method does the job of writing the output to the console window. The method itself looks like this:

```
private void WriteMetricsToConsole(PrintPageEventArgs ev)
{
    Graphics g = ev.Graphics;
    Console.WriteLine("Information about the printer:");
    Console.WriteLine("ev.PageSettings.PaperSize: " +
                      ev.PageSettings.PaperSize);
    Console.WriteLine("ev.PageSettings.PrinterResolution: " +
                      ev.PageSettings.PrinterResolution);
    Console.WriteLine("ev.PageSettings.PrinterSettings.LandscapeAngle: " +
                      ev.PageSettings.PrinterSettings.LandscapeAngle);
    Console.WriteLine("");
    Console.WriteLine("Information about the page: ");
    Console.WriteLine("ev.PageSettings.Bounds: " + ev.PageSettings.Bounds);
    Console.WriteLine("ev.PageBounds: " + ev.PageBounds);
    Console.WriteLine("ev.PageSettings.Margins: " +
                      ev.PageSettings.Margins);
    Console.WriteLine("ev.MarginBounds: " + ev.MarginBounds);
    Console.WriteLine("Horizontal resolution: " + g.DpiX);
    Console.WriteLine("Vertical resolution: " + g.DpiY);
    g.SetClip(ev.PageBounds);
    Console.WriteLine("g.VisibleClipBounds: " + g.VisibleClipBounds);
    SizeF drawingSurfaceSize = new SizeF(
        g.VisibleClipBounds.Width*g.DpiX/100,
        g.VisibleClipBounds.Height*g.DpiY/100);
    Console.WriteLine("Drawing Surface Size in Pixels: " +
                      drawingSurfaceSize);
}
```

Note Keep in mind that, in this example, we're accessing the settings within the `PrintPage` event handler only because it's convenient to do so for demonstration purposes. If you used this example to print a 20-page document, this event handler would write the same information to the console 20 times! If you were writing production code, you would probably access information about metrics from other places; for example, within the `Print` or `Print Preview` menu event handlers, using the `PrintDocument`'s `DefaultPageSettings` and `PrinterSettings` properties.

Note that there is an issue with the `VisibleClipBounds` property of the `Graphics` class. This property is the intersection between the clipping region of the drawing surface and the printable area of the printer. With some printers, if the clipping region has not previously been set, this property does not contain a valid value. If you come across this problem, you can work around it by setting the clipping region to `PageBounds` before accessing this property, as follows:

```
g.SetClip(ev.PageBounds);  
Console.WriteLine("g.VisibleClipBounds: " + g.VisibleClipBounds);
```

To see the output, run the example and select `Print` or `Print Preview`. Then inspect the console window, which reveals the following file contents:

Information about the printer:

```
ev.PageSettings.PaperSize: [PaperSize A4 Kind=A4 Height=1169 Width=827]  
ev.PageSettings.PrinterResolution: [PrinterResolution X=600 Y=600]  
ev.PageSettings.PrinterSettings.LandscapeAngle: 90
```

Information about the page:

```
ev.PageSettings.Bounds: {X=0,Y=0,Width=827,Height=1169}  
ev.PageBounds: {X=0,Y=0,Width=827,Height=1169}  
ev.PageSettings.Margins: [Margins Left=100 Right=100 Top=100 Bottom=100]  
ev.MarginBounds: {X=100,Y=100,Width=627,Height=969}  
Horizontal resolution: 600  
Vertical resolution: 600  
g.VisibleClipBounds: {X=0,Y=0,Width=827,Height=1169}  
Drawing Surface Size in Pixels: {Width=4962, Height=7014}
```

You may find that your printer and setup generate different results. It's instructive to use the `Page Setup` dialog box to change some settings, and then check the print preview again to see how the output changes. For example, try changing the printer orientation from portrait to landscape mode, and see the changes in the `ev.PageSettings.Bounds` values. Also, try changing the margins, and look for the effects in the `ev.PageSettings.Margins` and `ev.MarginBounds` values.

Controlling the Printer Drawing Surface

In this section, we'll examine how applications can programmatically control characteristics of the drawing surface through the values of some of the printing metrics. For example, you could perform any of the following actions programmatically:

- Change the resolution of the printer.
- Set the print quality to draft mode.
- Force the printer to print in landscape orientation.
- Force the printer to print in black and white.

All of these metric-changing activities involve writing an event handler for the `QueryPageSettings` event. A `QueryPageSettings` event is raised before each `PrintPage` event is raised.

The `QueryPageSettings` event handler expects two arguments. One is a `QueryPageSettingsEventArgs` object, which contains (as one of its properties) a `PageSettings` object. You can change the properties of this object within the `QueryPageSettings` event handler. When you do so, the values will be reflected in the subsequent `PrintPage` event, both in the values of its `PrintPageEventArgs.PageSettings` property and in the page that it prints.

Note that you cannot simply change the `PageSettings` in the `PrintPage` event. By that time, it is too

late—any changes made in the `PrintPage` event will have no affect on the printed page. While this sounds restrictive, it does make sense. The module that raises these events needs to construct the appropriate `Graphics` object before raising the `PrintPage` event, with appropriate resolutions, page bounds, and so on. This is why the `QueryPageSettings` event is raised *before* the `PrintPage` event, offering you a chance to allow the .NET Framework to prepare for the `PrintPage` event.

Any changes that you make to the `PageSettings` from within the `QueryPageSettings` event are effective only for the *next single* page to be printed. If you want to make changes that apply over subsequent pages, you need to apply the changes again in subsequent `QueryPageSettings` events. Alternatively, you can make changes that apply to the entire print job, as you'll see in the "[Changing Metrics for the Entire Print Job](#)" section later in this chapter.

PRINT SETTING PRECEDENCE AND USER EXPECTATIONS

In general, printing metrics and printer settings are determined in one of three ways:

- Values set by the user
- Values set programmatically
- System default values

The order of this list is significant, because it shows the expected order of priority of the different "parties" that influence these values. In other words, in normal circumstances, any values set programmatically by the application will take precedence over system default values. However, any values set by the user (using the Page Setup dialog box or the Print dialog box) take precedence over both.

In your application, you might choose to set certain metrics, if you know that the default values are not appropriate for your application. For example, if you know that your printout will look silly in portrait mode, you can programmatically override the system default of `Portrait` by setting `Landscape` mode. Subsequently, once your application has displayed a Page Setup dialog box, the user has a chance to override the default and programmatic settings that were in place when the application started.

When the user selects OK in the Page Setup dialog box, the settings and the metrics specified by the user are stored in the `PageSettings` object. Even when that has happened, it is still possible to override those user-specified settings in your code (as you'll see in an example shortly). However, it's not always a good idea to override your user's custom settings, because that will cause your application to behave counter to your user's expectations and will probably annoy him. If you believe that some option the user might choose is likely to be inappropriate for your application, it's far better to write code that displays a *warning* inviting the user to change his preferences, but then to do what the user wants.

This kind of design decision comes down to understanding your user's expectations and the requirements of the application.

Changing the Printer Resolution

Suppose that we are writing an application in which, for some reason, we must force the print quality to the lowest level, regardless of the print quality that the user specified. Each printer has a list of printer resolutions that are valid for that printer, and when we set the `PrinterResolution` property of the `PageSettings` object, we can set it to only one of the values in the list.

First, let's take a look at the available resolutions for a given printer. You can find out the available printer resolutions by querying the `QueryPageSettingsEventArgs` object's

`PageSettings.PrinterSettings.PrinterResolutions` collection. You can do this by using something as simple as a `foreach` loop. To see how this works, you can try this sample code, which displays output to a console window:

```
protected void QueryPageSettingsEventHandler( Object obj,
    QueryPageSettingsEventArgs e )
{
    ...
    foreach (PrinterResolution pr in
```

```

        e.PageSettings.PrinterSettings.PrinterResolutions)
    {
        Console.WriteLine(pr);
    }
}

```

If you run this fragment of code against a particular printer, you will get output to the console window that would look like this:

```

[PrinterResolution High]
[PrinterResolution Medium]
[PrinterResolution Low]
[PrinterResolution Draft]
[PrinterResolution X=200 Y=200]
[PrinterResolution X=300 Y=300]
[PrinterResolution X=600 Y=600]

```

In order to change the printer resolution, we need to loop through the collection of possible settings, find the one that we want, and then set the `PageSettings.PrinterResolution` property to our desired resolution. As mentioned earlier, the `PrinterResolution` class has a property called `Kind`, which represents the resolution of the printer and is set to one of the values in the `PrinterResolutionKind` enumeration (whose names are self-explanatory): `Custom`, `Draft`, `High`, `Low`, or `Medium`.

As we loop through the available printer resolutions, we'll test the `Kind` property against the appropriate member in the `QueryPageSettings` event handler, and hence set the printer resolution to the desired option.

To try this out, we'll build the `PrintingMetricsExample2` application. This application is exactly like the `PrintingMetricsExample` application, but with a couple of changes. First, we need a `QueryPageSettings` event handler like this:

```

protected void QueryPageSettingsEventHandler( Object obj,
    QueryPageSettingsEventArgs e )
{
    foreach (PrinterResolution pr in
        e.PageSettings.PrinterSettings.PrinterResolutions)
    {
        if (pr.Kind == PrinterResolutionKind.Low)
            e.PageSettings.PrinterResolution = pr;
        break;
    }
}

```

This event handler loops through the different printer resolutions until it finds one that is classified as a low-resolution setting, and then it sets the resolution to this value.

Of course, we need to modify the event handlers for the menu items for printing and previewing, to register the `QueryPageSettings` event handler in the `PrintDocument` object:

```

private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    try
    {
        PrintDocument pd = new PrintDocument();
        pd.QueryPageSettings += new QueryPageSettingsEventHandler(
            this.QueryPageSettingsEventHandler );
        pd.PrintPage +=
            new PrintPageEventHandler( this.PrintPageHandler );
    }
}

```

We also make the same change to the `menuFilePreview_Click` event handler.

Compile and run the application, and you should see the change in the printer resolution. Moreover, you can check the printer metrics output that is sent to the console window. You'll see the change reflected in the DPI settings as retrieved from the `Graphics` object. On my machine, doing this yielded the following, with the changes highlighted:

Information about the printer:

```
ev.PageSettings.PaperSize: [PaperSize A4 Kind=A4 Height=1169 Width=827]
```

```
ev.PageSettings.PrinterResolution:[PrinterResolution X=200 Y=200]
```

```
ev.PageSettings.PrinterSettings.LandscapeAngle: 90
```

Information about the page:

```
ev.PageSettings.Bounds: {X=0,Y=0,Width=827,Height=1169}
```

```
ev.PageBounds: {X=0,Y=0,Width=827,Height=1169}
```

```
ev.PageSettings.Margins: [Margins Left=100 Right=100 Top=100 Bottom=100]
```

```
ev.MarginBounds: {X=100,Y=100,Width=627,Height=969}
```

Horizontal resolution:200

Vertical resolution:200

```
g.VisibleClipBounds: {X=0,Y=0,Width=827,Height=1169}
```

```
Drawing Surface Size in Pixels: {Width=4962, Height=7014}
```

Forcing Page Orientation

You can force page orientation to either landscape or portrait mode by using the following code in the `QueryPageSettings` event handler:

```
protected void QueryPageSettingsEventHandler( Object obj,
    QueryPageSettingsEventArgs e )
{
    ...
    e.PageSettings.Landscape = true;
}
```

Forcing Color Depth

You can test for color depth using the `PageSettings.Color` property. There are many situations in which the `PageSettings.Color` setting is worth investigating. For example, if your application is one that draws bar charts that make use of colors, then it's a good idea to check the `PageSettings.Color` property to see if the printer supports color. If it doesn't, then you can get around the problem by employing hatch brushes instead of color brushes to paint the bar chart.

In the following code, we use the `QueryPageSettings` event handler to *force* the output to be in black and white:

```
protected void QueryPageSettingsEventHandler( Object obj,
    QueryPageSettingsEventArgs e )
{
    ...
    e.PageSettings.Color = false;
}
```

Changing Metrics for the Entire Print Job

As noted earlier, changes made within the `QueryPageSettings` event handler are on a page-by-page basis. In other words, the changes we made in the previous example have an effect on *only* the page printed immediately after that `QueryPageSettings` event.

It's also possible to make changes that affect the *entire* print job. You do this by making the property assignments within the `Click` event handler for the Print or Print Preview menu items, after you've created the `PrintDocument` object. For example, in the `menuFilePrint_Click` event handler, you can modify the `DefaultPageSettings` property, as follows:

```
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    try
    {
        PrintDocument pd = new PrintDocument();
        pd.PrintPage +=
            new PrintPageEventHandler( this.PrintPageEventHandler );

        // If the user has set page settings, then set the property
        // in the print document.
    }
```

```
if ( this.storedPageSettings != null )
    pd.DefaultPageSettings = this.storedPageSettings;

// Put document-level settings here
pd.DefaultPageSettings.Landscape = true;

PrintDialog dlg = new PrintDialog();
...
```

This will cause the entire document to be printed in landscape mode.

Note that any changes that you make to the `DefaultPageSettings` property affect *only* the current document being printed. These changes *don't* have any effect on the default settings for the printer.

Where the User Changes the Metrics Settings

As mentioned earlier, the various printing metrics can be stored in several places. To some extent, this reflects the fact that, traditionally, the user has been able to set different metrics in different places—some in the Page Setup dialog box, and others in the Print dialog box. In this section, we'll quickly review which settings can be set where using the Microsoft-supplied dialog boxes.

Settings in the Page Setup Dialog Box

As you've seen, the `Click` event handler for the Page Setup menu item puts up the Page Setup dialog box. The dialog box itself is represented by a `PageSetupDialog` object:

```
PageSetupDialog psDlg = new PageSetupDialog();

// Create a PageSettings object if never before created
if ( this.storedPageSettings == null )
    this.storedPageSettings = new PageSettings();

// Put up the dialog
psDlg.PageSettings = this.storedPageSettings;
psDlg.ShowDialog();
```

In this dialog box, the user can set the paper size, paper source, print orientation, and margins. Changes to any of these settings are reflected in the `PageSettings` object. Changes are made to the `PageSettings` object to which the `QueryPageSettings` event handler has access, where you can make additional changes to it. In addition, changes are reflected in the `PageSettings` object to which the `PrintPage` event handler has access.

Settings in the Print Dialog Box

The `Click` event handler for the Print menu item puts up the Print dialog box. This dialog box is represented by a `PrintDialog` object:

```
PrintDialog dlg = new PrintDialog();
dlg.Document = pd;
DialogResult result = dlg.ShowDialog();
if (result == System.Windows.Forms.DialogResult.OK)
    pd.Print();
```

If the user clicks the Properties button for the printer, she can set the paper size, graphics resolution, and print orientation. Again, changes to these settings are reflected in the `PageSettings` object.

 Previous

Next 

 Previous

Next 

Summary

This chapter covered the basics of how to make your Windows Forms applications support printing capabilities. In general, user expectation is for three main activities: page setup, printing, and print previewing. In many applications, we represent these three activities in the form of three standard dialog

boxes. GDI+ provides objects that represent these three dialog boxes (`PageSetupDialog`, `PrintDialog`, and `PrintPreviewDialog`), as well as a system of events and event handlers that you can use to react when the user selects these dialog boxes in your application's menu.

You saw that the general principles for drawing to a printer or print preview are the same as for drawing to the screen, to the extent that you control what is printed by calling the appropriate methods on a `Graphics` object, and those methods remain the same. You still use such favorites as `Graphics.DrawString` and `Graphics.DrawRectangle`. But at a deeper level, there are many differences between the screen and a printer. Most notably, with the screen, you might control the portion of the document that is displayed by using scroll bars in conjunction with the `Graphics.TranslateTransform`. Printers don't have scroll bars; instead, you print each page separately. Also, whereas the `Paint` event is raised by the operating system whenever it detects that something needs to be redrawn, printing is done in response to a specific request by the user, normally after displaying a `Print` dialog box. Therefore, the sequence of events when printing is slightly different.

There are some other special considerations when drawing to a printed page drawing surface. For example, you must deal with things like the printer's capabilities (in terms of resolution and color) and the physical setup of the page to be printed. The default system settings can be overridden in two ways: either programmatically in your application (using properties of the `PageSettings` and `PrinterSettings` objects that represent the print metrics) or through the user's custom settings, invoked via the `Print` and `Page Setup` dialog boxes. You can design your application in such a way that programmatic settings (chosen at design-time) override user settings (chosen at runtime), but this tends to go against the grain of user expectation. Ultimately, this issue is one of application design.

Another difference relates to resolution. Printers tend to have much higher resolution than screens, and that means that you need to consider your unit of measurement carefully. On a 96 DPI screen, a 100-pixel line is around 1 inch in length; on a 600 DPI printer, the same line is less than 1/6 inch! One way round this is to set the `Graphics.GraphicsUnit` property to `GraphicsUnit.Inch`, and then draw all graphics in terms of inches. This helps to ensure a consistency of appearance between the screen and printer, and gets you closer to achieving a situation in which you have a single set of code that draws what you want on both the screen and the printed page.

We covered the printing programming interface in detail, including how to create a `PrintDocument` object and implement a `PrintPage` event handler for it. When you want to modify the characteristics of the print document as a whole, you can change the `DefaultPageSettings` property of the `PrintDocument` object. When you want to change the drawing surface characteristics of an individual page, you can implement the `QueryPageSettings` event.

There are a large number of metrics that you can get about the printer and each printed page. You saw how to get these metrics. This allows you to write graphical code that takes the page orientation, color depth, and other metrics into account.

In the [next chapter](#), we are going to take a detour from our discussions of the mechanics of GDI+, and explore a technique that makes it easier to write graphics code.

 Previous

Next 

 PreviousNext 

Chapter 10: An Alternative Coordinate System

Overview

[Chapter 2](#) described how your drawing surface works in terms of rows and columns of pixels. You describe individual pixels in terms of a set of numbered gridlines, which run horizontally and vertically through the centers of the pixels. Thus, if you choose any vertical gridline and any horizontal gridline, they cross in exactly one point, and that point is the center of a pixel. Having established this grid coordinate system, we looked at various drawing and filling operations and saw how GDI+ interprets these drawing operations in terms of this coordinate system (and with consideration of other things like pen width, too).

In [Chapter 3](#), we came across a couple interesting effects related to this coordinate system. For example, at the pixel level, you must think about *drawing* and *filling* operations quite differently. When you use `DrawRectangle` to draw a square whose side length is, say, 100 pixels, the square drawn on the drawing surface has a side length of 100 pixels *plus* the width of the pen. Yet, if you use `FillRectangle` to fill the same square, the result is *exactly* 100 pixels square. Also, when you fill a square rectangle with integer pixel coordinates, with anti-aliasing switched on, GDI+ interprets this not as a whole number of pixels, but in terms of half-shaded pixels at the edges. This is in direct contrast to the behavior when you're dealing with regions, clipping, and invalidation, as you saw in [Chapters 6](#) and [7](#). A region is composed of *whole* pixels. It's as if the region were drawn with a grid whose gridlines *separate* adjacent rows and columns of pixels, rather than running through the *centers* of the pixels.

When you're using both functions in the same graphic, and you want to control the image carefully at the pixel level, inconsistent behavior like this makes your job much more difficult. When you're working on a screen resolution of around 96 DPI, you want to be able to control the exact color of each pixel as closely as possible. But the same dimensions can produce slightly different sizes and effects, depending on whether you're drawing it, filling it with anti-aliasing, filling it without anti-aliasing, or making a region out of it—and that makes it much harder to maintain control at the pixel level.

I have written custom controls for all sorts of platforms: for versions of Microsoft Windows from version 2.0, for the Macintosh, for OS/2, for X/Motif (and even OpenLook), and other windowing systems too obscure to mention. Most of these windowing systems define their own semantics for the behavior of drawing operations, and some even change behavior from version to version. In some cases, I wrote some custom controls that were portable between all of the above windowing systems, with the drawing operations abstracted out, so that the bulk of the code was the same from platform to platform.

However, the variety in drawing operation behavior between different platforms was problematic. Out of sheer necessity, I devised an alternative coordinate system model. I wrapped all of the drawing operations on all of the platforms with this alternative coordinate system model, and this allowed me to write many custom controls that were portable between *all* of the platforms. Along the way, I found that it was much easier to program with my new coordinate system model. I've shared my coordinate system model with many other programmers, with some positive feedback. Now, even if I don't intend to port a custom control to another platform, I still wrap the graphical operations so that I can use this coordinate system, just because it is easier to program.

I call this coordinate system the *Outline Model coordinate system*. It is implemented in a single class, called the `GraphicsOM` class. It is easy to use, and it is particularly useful when writing custom controls in which you would like to have precise control over which pixels are drawn. Almost all screens have a resolution of less than 123 DPI, and most have a resolution of around 96 DPI. With screens of this level of resolution, control of individual pixels is very important to the overall effect you're trying to create.

This chapter describes how to use this alternative coordinate system to overcome the problems of the default GDI+ coordinate system. It covers the following topics:

- The problems with the default coordinate system in GDI+
- The Outline Model coordinate system—how to use it and how it solves the problems
- Some 3D raised and inset effects that are included in the `GraphicsOM` class library

 PreviousNext 

[Previous](#)[Next](#)

Defining the Problem

Here's the problem: when you define a shape or line of a given pixel size in your code, the *actual* pixel size of that shape or line on the drawing surface depends on the operation you're applying. As a direct result, it's much harder to control the color of your graphic at the pixel level. Consequently, it's difficult to arrange shapes and outlines in such a way that they meet exactly without overshooting or falling short. This ultimately means that (at best) your graphics look untidy, and (at worst) 3D effects and other visual effects are spoiled.

Let's review some examples from recent chapters. [Figure 10-1](#) illustrates some operations that attempt to draw and fill lines and squares that involve lengths of 3 pixels. Here's what happens with the diagrams on the left side of [Figure 10-1](#), which are all drawn with a pen of width 1 pixel:

- The horizontal line is defined as starting at the center of (0, 0) and ending at the center of (3, 0). Therefore, by definition, it is 3 pixels in length. However, the pen width is exactly 1 pixel, and so by operation, each of the four pixels between (0, 0) and (3, 0) inclusive is colored black. The result is a line that measures 4 pixels from end to end.
- The diagonal line reaches from (0, 0) to (3, 3). By a similar argument, it stretches through a full four pixels, not three.
- The bottom-left diagram in [Figure 10-1](#) is the outline of a square of a height and width of 3 pixels. By a similar argument, each of its sides is represented by a *four-pixel* block.

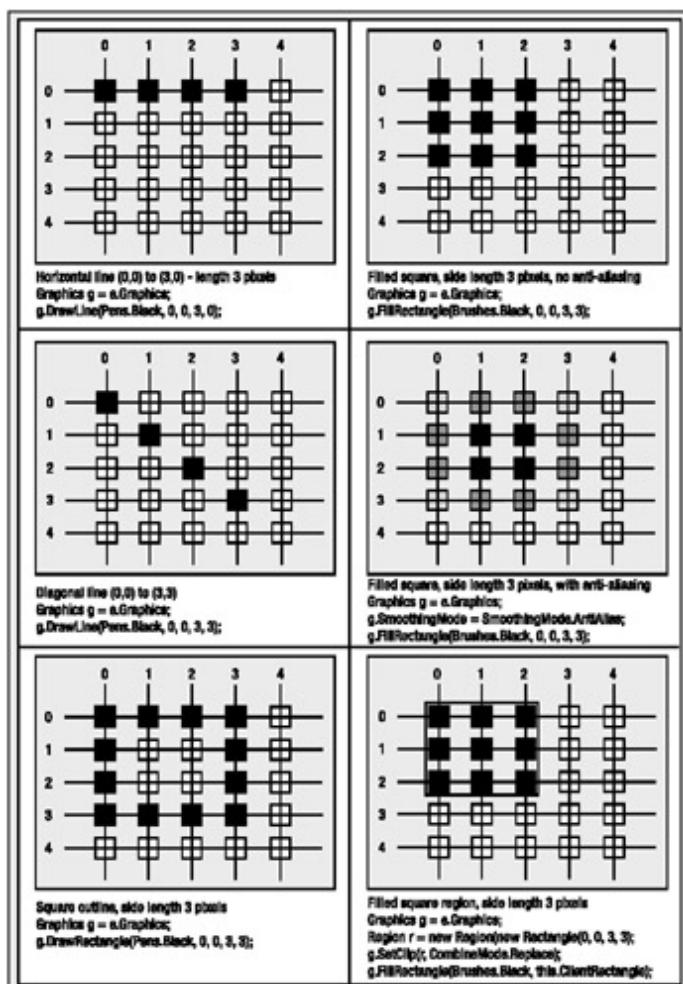


Figure 10-1: Varieties of behavior

Now compare the behavior of the diagrams on the left with the variable behavior demonstrated by the

examples on the right side of [Figure 10-1](#):

- The first is another square of a height and width of 3 pixels, but this time it's filled (with no anti-aliasing). According to the `FillRectangle` request, the top-left corner of the square should be at (0, 0). However, with anti-aliasing disabled, GDI+ is permitted to draw only whole pixels, so this isn't possible. The result is a square of identical size, but which has been offset by a half pixel upwards and left (because the "no anti-aliasing" algorithm is such that the pixels that hang over the left edge and above the top edge are filled, but the pixels that hang over the bottom edge and over the right edge are not filled). Thus, the result is a filled-in square that's *exactly* three pixels in each direction.
- The second is *another* filled square of a height and width 3 of pixels, but this time anti-aliasing is switched on. The size of this square is the same as the one above it, and, in fact, it represents the `FillRectangle` request in that it tries to put the top-left corner of the square at (0, 0). To achieve this effect, the edge pixels of the square are half-shaded, as described in [Chapter 2](#).
- The bottom-right diagram in [Figure 10-1](#) shows the drawing surface clipped to a 3X3 clipping region, and then filled. The clipping rectangle has three pixels on a side, and so a total of nine pixels are affected. In fact, it's similar to the square filled without anti-aliasing.

The six different operations shown in [Figure 10-1](#) result in three different types of behavior. In the three `Draw...` operations (on the left), the pen width has an effect, so that you actually get a line based on the width of four pixels, not three. Two of the remaining three operations affect exactly the same pixels (the square filled without anti-aliasing and the clipping region). The square filled *with* anti-aliasing produced an apparent half-pixel shift when compared with the square filled *without* anti-aliasing, and the edge pixels have anti-aliasing artifacts (that is, they are shaded). In fact, the inconsistency can be broken down into two distinct issues.

First, when you request a `Draw...` operation, GDI+'s interpretation of the operation results in a graphic that is larger than the dimensions you've specified. This is because the pen width is nonzero, and so influences the length of lines drawn (as well as the width). Thus, a 3-pixel line drawn with a 1-pixel pen results in a line of length $3+1=4$ pixels; a 10-pixel line drawn with a 5-pixel pen results in a line of length $10+5=15$ pixels. It might be preferable to have a system that represents the pen width in the width of the graphic, but not in the length.

Second, some of these actions can be described naturally by thinking of gridlines that run through the center of the pixels (that's especially true of `Fill...` operations with anti-aliasing). Others are more natural when you consider them in terms of *whole* pixels, described using gridlines that separate the rows and columns of pixels (for example, defining and using a region). Others (such as the `Draw...` operations) can actually be described either way. It would be preferable if we used a system of gridlines that described all these operations naturally.

Now that we've identified the problem, let's describe how to resolve it.

[Previous](#)

[Next](#)

[Previous](#)

[Next](#)

Introducing the Outline Model Coordinate System

Let's take a look at an alternative approach that tackles these problems. We execute a two-pronged attack on our drawing ideas.

First, we make a subtle change to the way we think of our coordinate system. Specifically, we'll move the gridlines that we use to describe the positions of the pixels. Thus, instead of running the gridlines through the *centers* of the rows and columns of pixels, we'll have those gridlines running *between* the pixels, as shown in [Figure 10-2](#).

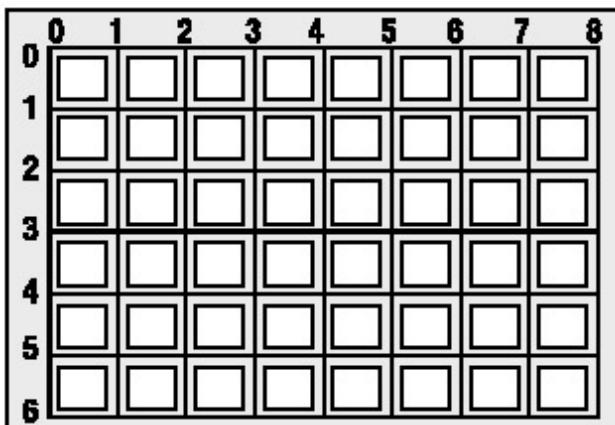


Figure 10-2: Gridlines running between the pixels

In this system, the expression (x, y) describes the pixel immediately to the *right* of the x th vertical gridline and immediately *below* the y th horizontal gridline. By contrast, in the default system (x, y) describes the pixel *centered* on the x th vertical gridline and y th horizontal gridline.

We'll use this slightly revised coordinate system to describe the positions of individual pixels and sets of pixels to which we want to draw. When we perform our drawing and filling operations using this new coordinate system, we will need GDI+ to interpret our instructions slightly differently. To do that, we'll use a custom class called `GraphicsOM`, which performs the necessary interpretation between our new coordinate system and the default one, and then uses the standard GDI+ `Graphics` class to draw the translated instructions to the drawing surface.

Second, our `GraphicsOM` class's `Draw` methods will also allow us to draw lines and shapes whose actual dimensions depend only on the dimensions of the lines and shapes given. They will be *independent* of the pen width. We accomplish this by having the pen fall inside of our shape, instead of straddling the outline.

The `GraphicsOM` class is a simple custom class that replaces some of the functionality of GDI+'s `Graphics` class. The `GraphicsOM` class consists of a number of methods (such as `DrawRectangle`, `DrawLine`, and a few others) that you can use in a very similar way to the similarly named methods of the `Graphics` class. However, they interpret your drawing operations slightly differently, in that they take account of the change in coordinate system, and they draw lines whose length is independent of pen width.

The `GraphicsOM` class includes all of the code for drawing horizontal lines, vertical lines, diagonal lines, and filled rectangles, as well as all methods that draw raised and inset rectangles and lines. Most of these methods don't do a lot; they just alter coordinates, and then call their corresponding method in the `Graphics` class.

We won't cover the source code for the `GraphicsOM` class here. It is available from the Downloads section of the Apress web site (along with the other code presented in this book) at www.apress.com, if you would like to examine it or use it in your own custom controls and applications. However, I will explain how to compile the class into a usable class library, and how to employ it in your code.

Building the `GraphicsOM` DLL

You don't need to build the `GraphicsOM` class library. If you prefer, you can use the `Apress.GraphicsOM.dll` file that is already compiled and ready to use (available from the Apress web site).

However, if you would prefer to compile the class library for yourself, here's how you can do it in Visual Studio .NET:

1. Create a new Visual C# Class Library project and give it a name (perhaps `Apress.GraphicsOutlineModel`). Place a copy of the `GraphicsOM.cs` file (available as part of the source code for this book) in the directory for the new project.
2. In the Solution Explorer window, delete the existing `Class1.cs` file. Then right-click the project in

the Solution Explorer window and select Add ➔ Add Existing Item. Select the `GraphicsOM.cs` file in the dialog box, and add it to your project.

3. In the Solution Explorer window, right-click References and select Add Reference. Select `System.Drawing.dll` in the list box. Click the Select button, and then click the OK button. Repeat the process and add a reference to `System.Windows.Forms.dll`.
4. Build the DLL by selecting Build ➔ Build Solution. Now look in the `/bin/debug` directory of your project. You will see the `Apress.GraphicsOutlineModel.dll` file that you've just created by compiling the code. That's it. You can close this project.

Now you're ready to test the `GraphicsOM` class by using it in some sample code.

Using the `GraphicsOM` Class

The methods of the `GraphicsOM` class work in just the same way as their namesake methods in the `Graphics` class. Using these methods is quite easy.

In order to use the `GraphicsOM` class in drawing code, you first need to add a reference to the `Apress.GraphicsOutlineModel.dll` class library file (in the Solution Explorer window, right-click the References node, select Add Reference, click the Browse button, navigate to the `Apress.GraphicsOutlineModel.dll` file, and select it).

The class belongs to a namespace called `Apress.GraphicsOutlineModel`. You need to add a `using` statement to indicate that you're using code from that namespace:

```
using Apress.GraphicsOutlineModel;
```

Now you can use the `GraphicsOM` class in your painting code. You write a `Paint` event handler in the normal fashion. Inside the event handler, you declare and instantiate an instance of the `GraphicsOM` class, passing the `Graphics` object to the constructor:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsOM gOM = new GraphicsOM(g);
    ...
}
```

You can set properties such as clipping and smoothing mode using the `Graphics` class, but call drawing and filling operations using the `GraphicsOM` class. The `GraphicsOM` object will interpret your instructions and draw them correctly on the screen. Let's take a look at some examples.

Regions and Clipping Operations

The `GraphicsOM` class uses identical semantics to the `Graphics` class when it comes to regions and clipping, and so its behavior does not change. Thus, a region clipped using the `GraphicsOM` class looks just like one that is clipped using the `Graphics` class, as shown in [Figure 10-3](#).

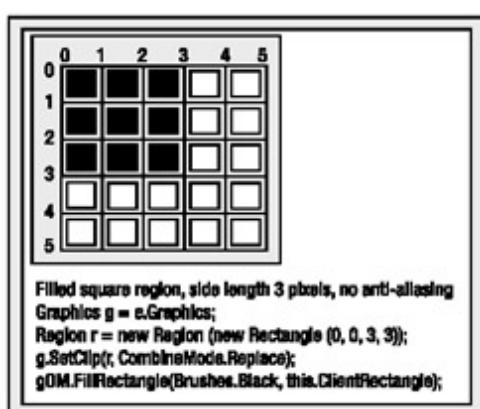
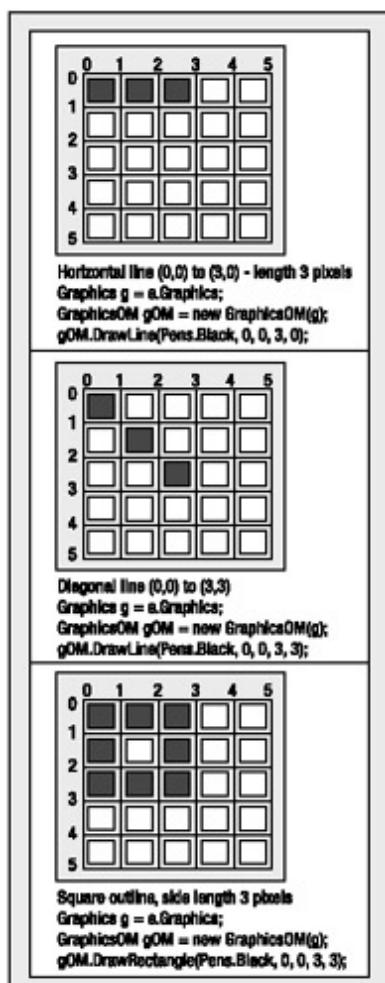


Figure 10-3: Filled square using GraphicsOM

Drawing Operations

Now let's revisit the `DrawLine` and `DrawRectangle` examples discussed earlier in the chapter (see [Figure 10-1](#)). The three examples shown in [Figure 10-4](#) use `DrawLine` and `DrawRectangle` method calls that specify the same dimensions as before, but now the operations are requested through the `GraphicsOM` class, instead of the `Graphics` class. As a result, the resulting lines and shapes more accurately reflect the lengths that were intended:

- When drawing a horizontal line with a pixel width of 1, you define that the pixels drawn fall just *below* your grid line. In the example in [Figure 10-4](#), the line drawn from point (0, 0) to point (3, 0) draws three pixels just below the zeroth horizontal gridline. In a similar fashion, a vertical line drawn, say, from (10, 10) to (10, 15) will consist of five (not six) pixels immediately to the right of the tenth vertical gridline.
- When drawing a diagonal line, you use a rectangle whose diagonally opposite corners are the beginning and ending points of the line. The two pixels *just inside* these corners of the rectangle are affected by the drawing operation, and a straight (or as straight as possible) line is drawn between those two pixels. The example in [Figure 10-4](#) shows a diagonal line drawn from pixel (0, 0) to pixel (3, 3). The line is contained *between* the zeroth and third gridlines in each direction.
- When drawing a rectangle with any pixel width, the pixels drawn will, by the definition of `GraphicsOM`, fall just inside the gridlines that make up the rectangle. For example, if you use a 1-pixel pen to draw a rectangle whose upper-left corner is at (1, 1), and whose width and height are defined to be 3 pixels, `GraphicsOM` draws the rectangle shown at the bottom of [Figure 10-4](#).

**Figure 10-4:** GraphicsOM drawing operations

Thus, in each of these cases, if you were to set the clipping region of your drawing surface to a given rectangle, and then use the same dimensions to draw a rectangle or diagonal line with

`GraphicsOM.DrawRectangle` or `GraphicsOM.DrawLine`, the drawn line or rectangle would fall just inside the clipping rectangle. This is one advantage that `GraphicsOM` offers over the `Graphics` class: the `Draw...` operations are more closely related to the region operations.

Fill Operations

Next, let's look at the `Fill...` operations, with and without anti-aliasing. In fact, when you use `GraphicsOM.FillRectangle` to fill a rectangle with integer dimensions, it doesn't matter whether you use anti-aliasing or not. The result is the same. For example, [Figure 10-5](#) shows what happens when you fill a rectangle whose top-left corner is at (0, 0) and whose width and height are both defined to be 3 pixels. The one on the right is anti-aliased.

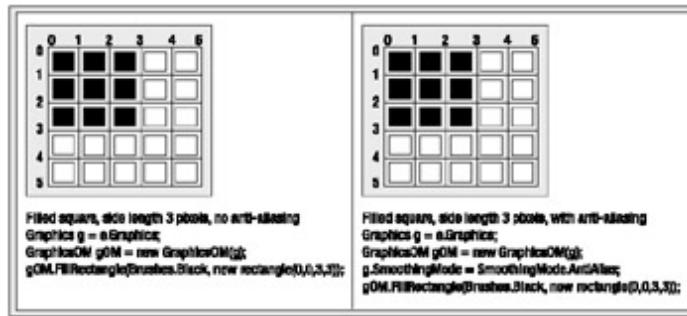


Figure 10-5: Filled rectangles drawn with `GraphicsOM`

Both of the rectangles shown in [Figure 10-5](#) are identical to the result you get when you fill the same rectangle using `Graphics.FillRectangle` with anti-aliasing turned off. However, this doesn't mean that `GraphicsOM` disables anti-aliasing altogether! Take a look at the two examples in [Figure 10-6](#). They show the results when you draw a 3.5-pixel square (using a `RectangleF`), with anti-aliasing switched on. The left example was drawn using `Graphics.DrawRectangle`, and the one on the right was drawn with `GraphicsOM.DrawRectangle`.

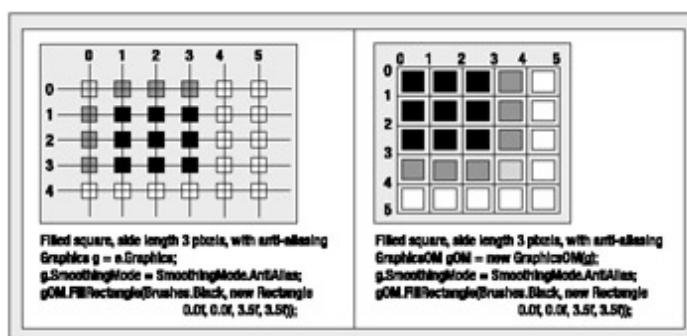


Figure 10-6: Anti-aliasing with `GraphicsOM`

In each example in [Figure 10-6](#), this is what you would expect. The anti-aliasing effect takes care of the fact that the square has noninteger dimensions. In both examples, the point (0, 0) is the top-left corner of the rectangle. In the coordinate system on the left, GDI+ represents this by anti-aliasing the (0, 0) pixel to a 25% gray shade. In the coordinate system on the right, the (0, 0) pixel is filled in black, and the rightmost and lowest pixels have been anti-aliased. More important, the system on the right is more compatible with the clipping region definition system.

DESIGN PATTERNS AND THE GRAPHICS OUTLINE MODEL

As I've said, my motivation for setting up the `GraphicsOM` class was prompted by two problems: that `Draw...` operations tended to use the length of the pen to determine the final length of a line or shape, and that the `Fill...` operations work differently from region operations for similar shapes.

Ideally, this solution would be implemented within the `Graphics` class itself. You would specify your choice of coordinate system by setting the `PageUnit` property of the `Graphics` class. If it had been possible, I would have implemented this model by deriving from the `Graphics` class. Unfortunately, this was not possible, because the `Graphics` class is sealed.

Instead, I've used a variation of a design pattern, known as the *decorator pattern*, to create a class with this functionality. In case you didn't know, *design patterns* are based on the idea that experienced programmers repeatedly come across similar types of problems, and that similar types of problems can be solved using similar solutions. This is also true in object-oriented design and programming. Programmers come across different problems that have similar characteristics, and find that they can apply a single solution to every one of those similar problems. The subject of design patterns is about identifying recurring problems and naming pattern solutions. As a result, it standardizes the approach that you take to solve a given type of problem. It gives programmers a better chance of applying the optimal solution to a problem, and it provides a lexicon to programmers so that they can communicate more rapidly about the design of class hierarchies.

The decorator pattern allows you to add functionality to a class without subclassing it. The idea here is that you make a set of methods with the same signature as the methods whose behavior you want to modify. When you instantiate the `GraphicsOM` class, you pass an instance of the `Graphics` class to the constructor, which gets held in a private member variable of the `GraphicsOM` object. When you call a method to perform a graphical operation, it collects the values passed in as parameters and modifies them to express your *desired* behavior in terms of the system used by the `Graphics` class, and then calls the corresponding method in the `Graphics` class to do the job.

If you would like to know more about the concept of design patterns, one good reference is *Design Patterns: Elements of Reusable Object-Oriented Software*(Addison-Wesley, ISBN 0-201-63361-2).

Pros and Cons of Using `GraphicsOM`

There are a number of advantages that the Outline Model coordinate system has over the default one in GDI+:

- When making a complicated custom control, it is convenient to divide the control into rectangles, and write methods that are responsible for drawing all the pixels within a rectangle but are not responsible for (and should not affect) any pixels outside that rectangle.
- You can pass integer coordinates to your methods, but you don't need to worry about whether anti-aliasing is turned on or off.
- There is a simple, direct relationship between the gridlines and the number of pixels affected by any drawing operation. For example, if you draw a rectangle whose left edge is at X=25, and whose right edge is at X=60, the resulting rectangle will be 60–25=35 pixels wide.
- Computation of clipping rectangles is made significantly easier, as they are expressed in exactly the same terms as your drawing operations. As I've mentioned, clipping doesn't work particularly well with the default GDI+ coordinate system.

However, the model also has a couple of disadvantages:

- It is not standard. Maintenance programmers will need to be trained in this alternative coordinate system.
- It may cause conflicts if it is used by some members of a programming group and not others. It may be desirable to standardize within a group or project, so that all developers use it or none do.

You need to use the Outline Model coordinate system only when you are drawing a number of lines and rectangles that must provide very precise coverage over a certain area of the form or window. It may not be appropriate if you are doing drawing where you will use anti-aliasing. You don't need to use it when drawing text.

There is no significant performance penalty associated with using the `GraphicsOM` class. The cost of creating and using the `GraphicsOM` class, relative to the costs of implementing the actual graphical operations and writing to the video card or display device, is minuscule.

Personally, I have found this approach to be a convenient and usable programming model that wears well with time. It yields maintainable programs.

[Previous](#)[Next](#)

Creating a 3D Effect

Since this chapter relates to the subject of drawing with exacting pixel precision, it seems reasonable to include a discussion of how you can create portions of your custom control that have a three-dimensional (raised or inset) appearance. It's common to see such effects in custom controls, and they are not difficult to create.

The `GraphicsOM` class includes a number of raised and inset effects. The raised effects are called `Ridge` and `Raised`, and the inset effects are called `Groove` and `Inset`. There are two depths for each style, as shown in [Table 10-1](#).

Table 10-1: GraphicsOM Raised and Inset Effects

Style	Depth 1	Depth 2
Ridge		
Groove		
Raised		
Inset		

As you can see, `Ridge` and `Groove` are visually inverse to one another, and `Raised` and `Inset` are also visually inverse to one another. In each case, light and dark colors create the 3D effect. The lighter colored areas give the effect that strong light is falling on an angled surface, while a darker shade gives the appearance of a surface that is angled away from the light and therefore in shadow. The heavily lit surfaces and shadowy surfaces combine to give the desired 3D effect, as long as you get the shades and lines right!

Note that in these effects, the light source always appears to be at the upper left. I've used this direction because Microsoft Windows uses the same position for its light source, and because the controls will be used in the Windows environment. If the light source were elsewhere (say, at the top right), then your controls would look very strange when viewed in Windows, where the light source for every other control is at the top left.

Ideally, you would implement this functionality using custom pens and brushes. However, this is not feasible using the .NET Framework. GDI+ was designed in such a way that you cannot derive from the `Pen` and `Brush` classes to add your own functionality. Instead, you'll find this functionality in the `GraphicsOM` class.

Using System Colors

When drawing using 3D effects, it's best to use the colors that are used elsewhere in the system for similar controls. Thus, if a user decides to change her system color scheme, your control will reflect the scheme. You can get these colors from the `System.Drawing.SystemColors` class. There are five properties in this class that are specifically intended to confer a 3D appearance in graphics, as described in [Table 10-2](#).

Table 10-2: The `SystemColors` Class Properties for 3D Effects

Property	Usage
<code>Control</code>	The face color
<code>ControlLight</code>	The light color of a 3D element
<code>ControlLightLight</code>	The highlight color of a 3D element
<code>ControlDark</code>	The shadow color of a 3D element

ControlDarkDark

The darkest color of a 3D element

[Figure 10-7](#) shows how these five colors are used in the Raised 3D graphic.

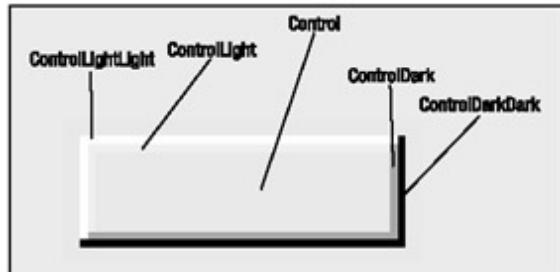


Figure 10-7: Use of the SystemColors properties

Drawing 3D Lines and Rectangles

The `GraphicsOM` class includes two overloaded methods that create these 3D effects: `Draw3DLine` and `Draw3DRectangle`. The way you use these methods is similar (though not identical) to the way you use the `Graphics` class's `DrawLine` and `DrawRectangle` methods. They differ in that you don't need to pass a `Pen` object to these functions, but you do need to specify whether your line or rectangle should have a `Groove` or `Ridge` effect.

Because these methods are part of the `GraphicsOM` class, their behavior is consistent with the other drawing methods in the class:

- When you draw a horizontal 3D line, it runs directly below the specified gridline.
- When you draw a vertical 3D line, it runs directly to the right of the specified gridline.
- When you draw a 3D rectangle, regardless of its style or depth, the rectangle is drawn just inside the gridlines that define its boundaries.

We'll cover a little of the structure here, but not all of the code. You're welcome to examine the source code for the `GraphicsOM` class, which is available as part of the downloadable code for this book (from the Downloads section of www.apress.com).

In the `GraphicsOM` source, there is an enumeration (called `ThreeDStyle`) that allows you to specify the style of your lines and rectangles:

```
public enum ThreeDStyle
{
    Groove,
    Inset,
    Raised,
    Ridge
}
```

There are two overloads for the `Draw3DLine` method:

```
public void Draw3DLine(p1, p2, style, depth);
public void Draw3DLine(x1, y1, x2, y2, style, depth);
```

Here, `p1` and `p2` are of type `Point`, `x1`, `y1`, `x2`, and `y2` are integers; `style` is one of the `ThreeDStyle` enumeration values; and `depth` is 1 or 2.

Here's an example of drawing 3D lines:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsOM gOM = new GraphicsOM(g);

    gOM.Draw3DLine(0, 10, 500, 10, ThreeDStyle.Ridge, 1);
```

```

gOM.DrawLine(0, 20, 500, 20, ThreeDStyle.Groove, 1);
gOM.DrawLine(0, 30, 500, 30, ThreeDStyle.Ridge, 2);
gOM.DrawLine(0, 40, 500, 40, ThreeDStyle.Groove, 2);
gOM.DrawLine(0, 50, 500, 50, ThreeDStyle.Raised, 1);
gOM.DrawLine(0, 60, 500, 60, ThreeDStyle.Inset, 1);
gOM.DrawLine(0, 70, 500, 70, ThreeDStyle.Raised, 2);
gOM.DrawLine(0, 80, 500, 80, ThreeDStyle.Inset, 2);
}
}

```

This code produces a number of lines that give various horizontal 3D effects, as illustrated in [Figure 10-8](#).

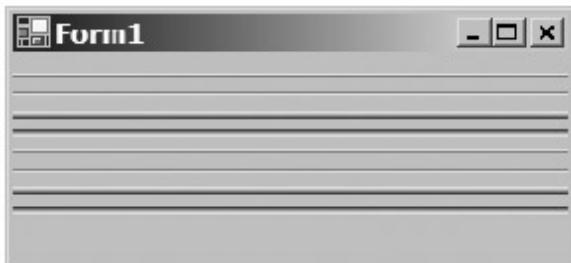


Figure 10-8: Various styles of 3D lines

There are also two overloads for the `Draw3DRectangle` method:

```

public void Draw3DRectangle(r, style, depth);
public void Draw3DRectangle(left, top, width, height, style, depth);

```

Here, `r` is a `Rectangle`. `left`, `top`, `width`, and `height` are integers that represent the position of a rectangle; `style` is one of the `ThreeDStyle` enumeration values; and `depth` is 1 or 2.

Here's an example of drawing 3D rectangles:

```

private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsOM gOM = new GraphicsOM(g);

    gOM.Draw3DRectangle(10, 10, 50, 20, ThreeDStyle.Ridge, 1);
    gOM.Draw3DRectangle(80, 10, 50, 20, ThreeDStyle.Groove, 1);
    gOM.Draw3DRectangle(10, 50, 50, 20, ThreeDStyle.Ridge, 2);

    gOM.Draw3DRectangle(80, 50, 50, 20, ThreeDStyle.Groove, 2);
    gOM.Draw3DRectangle(10, 90, 50, 20, ThreeDStyle.Raised, 1);
    gOM.Draw3DRectangle(80, 90, 50, 20, ThreeDStyle.Inset, 1);
    gOM.Draw3DRectangle(10, 130, 50, 20, ThreeDStyle.Raised, 2);
    gOM.Draw3DRectangle(80, 130, 50, 20, ThreeDStyle.Inset, 2);
}
}

```

This code produces a number of 3D rectangles, as shown in [Figure 10-9](#), which should give you an idea of how they work.



Figure 10-9: Various styles of 3D rectangles PreviousNext  PreviousNext 

Summary

In this chapter, we discussed the various ways in which coordinate systems that are used with drawing and clipping don't really work together in a natural way. You saw examples of several drawing operations using GDI+—specifically, `Draw...` and `Fill...` operations and operations involving regions. I pointed out how, even though the operations have similar coordinates, the parts of the drawing surface affected by these operations are rather different at the pixel level. When you're drawing to drawing surfaces that have a resolution of around 96 DPI, as most screens do, this level of detail is sufficient to spoil 3D and other effects, or simply make your controls look untidy.

We took a look at the `GraphicsOM` class, which implements an alternative coordinate system, called the Outline Model coordinate system. One of the advantages of this coordinate system is that if you pass identical coordinates to methods for drawing rectangles, filling rectangles, clipping, invalidation, and drawing lines, an identical set of pixels is affected in each case.

Finally, we looked at how the `GraphicsOM` class uses these techniques to draw rectangles and lines that have a raised or inset appearance, and how these operations have nearly identical behavior to the other `GraphicsOM` drawing methods. We didn't discuss the source code of the `GraphicsOM` class here, but it is available as part of the source code that accompanies this book, which is available from the Downloads section of the Apress web site (www.apress.com).

 PreviousNext 

 PreviousNext 

Chapter 11: Architecture and Design of Windows Forms Custom Controls

Overview

This chapter introduces the concept of the *custom control*—the real aim of this book. If you are reading this, you don't need to be reminded of the importance of being able to create custom controls that offer users features that are not directly available with the predefined Windows Forms controls supplied by Microsoft.

Whether you are building a Windows Forms rich-client application or an ASP.NET web application, the techniques for building components are identical. However, the techniques for building custom controls are quite a bit different. This chapter focuses on building custom controls for Windows Forms.

We will start off by reviewing some of the characteristics of custom controls and components in some detail. Then we will move on to the practical side and develop some custom controls. We will consider several aspects of custom controls:

- How to add properties to a custom control and inform Visual Studio .NET of default property values, so that the IDE can display properties in a way that makes it easy for other developers to reuse your control
- How controls should behave when they participate in the tabbing order of a form and receive or lose the focus, including how to correctly display keyboard UI cues
- How to generate your own events for a control
- How to derive a custom control from an existing Windows Forms control, allowing you to easily add custom functionality to standard controls
- How to build a custom control from several constituent controls, providing complex functionality that you can reuse in your applications

Once we have done all that, we will finish off by looking at some of the more theoretical considerations of control design, which you might want to consider when you are working on your own commercial controls.

 PreviousNext  PreviousNext 

Basics of .NET Controls

Most developers who are reading this book will already have a good understanding of both components and custom controls, which have been around in one form or another for quite a while. However, the .NET Framework adds some characteristics to components and custom controls that are worth discussing.

Component technology is very similar to object-oriented technology. In the .NET Framework, the two are closer than ever. However, there are aspects of component technology that are not part of traditional object-oriented programming (OOP), as you will learn in this section.

As you probably know, components and custom controls offer several advantages. Components are commonly used as a means to conceptually break an application into smaller pieces. They make software systems more modular and enable greater levels of code reuse. Components can be a convenient way to divide a large application among multiple programming groups, where each programming group is responsible for delivering a specific component or a set of components. Buying ready-made components and custom controls allows developers to use their time creating the custom aspects of their application, rather than handling more generic issues.

In addition, because .NET components have a very good mechanism for implementing properties, developers can use components and controls to implement a *declarative style* of programming. This is a particular programming style that increases programmer productivity and the robustness of the system, and decreases maintenance time. In brief, a declarative programming style is one where you focus on nonprocedural programming, rather than writing procedural code. Certainly, you will be writing procedural code, but you will be paying close attention to the nonprocedural portions of your application.

A declarative style of programming has such an advantage over the alternative that, for many developers, the idea of using a component without being able to bring up a Properties window to configure it would not even occur to them. At the end of this chapter, I will explain the generalization of declarative programming. It is a very valuable way to analyze and develop large software systems.

What is a Component?

Components are an implementation of certain abstract ideas, and it is helpful to understand these ideas. Understanding the intent helps you to design better components.

There is a fair amount of debate in the software industry as to exactly what a component *is* and what it *isn't*. Some technologies don't include all of the characteristics listed here, and proponents of those technologies argue that the characteristic should not be a required part of component technology. However, I think that all of these characteristics are important in the definition of a component.

Some aspects of components overlap aspects of OOP, but there are differences. In previous implementations of component technology, parts of OOP, such as inheritance, clearly were not part of the component technology, and parts of component technology, such as distributability, were not part of OOP. However, component technology in .NET now has all of the features of OOP. So, if components encompass all the benefits of OOP, why wouldn't we always build components? Component technology adds quite a bit of overhead, both in design, and in implementation. If you don't need the benefits of component technology, it is not a good idea to use that approach.

Note In the .NET Framework, any component that you build must implement the `System.ComponentModel.IComponent` interface or be derived from a class that implements it. Generally, you will not implement the `IComponent` interface yourself, but will instead derive directly or indirectly from the `Component` class. Note that `System.Windows.Forms.Form` derives indirectly from `Component`.

Encapsulation

A component encapsulates an abstraction or set of related functionality. In the same manner as OOP, a component packages data and procedures within it, hiding the implementation.

Interface

A component has a clear and specific programming interface (sometimes called a *contractually specified interface*). A component allows access to its data and behavior only through its programming interface. This is not the same as the user interface, which is a characteristic of a control.

Object-oriented developers working with a large class hierarchy often find themselves changing classes quite often. The ease with which you can do this and not break existing code is one of the strengths of OOP. In contrast, components implement a specific interface, and if the contract with the users of the component changes, you define a new interface. In other words, you don't change the external interface of a component so readily.

The interface to a component is made up of the following:

- **Properties:** Properties enable the user of the component to configure it in a declarative fashion (see the "[Designing Components and Custom Controls](#)" section later in this chapter for details).
- **Methods:** All component technologies have methods.
- **Events:** In today's event-driven world, components need to be able to both implement event handlers to receive events and generate events to be sent to the module using the component.

Some component technologies don't have all of these, but I consider all three to be vital.

Inheritance

Most component technologies before .NET did not include inheritance as a feature. However, it is very valuable to be able to declare a class that has a component as its base class. This allows you to do interesting things such as derive from an existing control and modify some part of its behavior while inheriting the remainder of its behavior.

Introspection

Introspection is a mechanism whereby users of the component can programmatically discover the interface to the component. This is vital to an IDE such as Visual Studio .NET, so that it can allow the developer to lay out the control, set properties, and implement event handlers.

Introspection is also used so that a program can discover new components and integrate the behavior of the components in the program. An example of this might be an image-editing program that allows for custom add-ins. However, the vast majority of the use of introspection is for integration into an IDE.

State

When you instantiate a component, it keeps state for your instance, and you can manipulate the state through the interface. You can implement stateless components, and often it is desirable to do so, but a good component technology allows components with and without state.

Persistence

When a component is used in an IDE, and the user has configured it in a certain way, there needs to be a mechanism to save the state of the component. In some technologies, this is implemented as object serialization. In other technologies, persistence is accomplished through automated procedural code parsing and generation technologies.

Pluggability

Separation of the interface from the implementation allows you to plug in one of a number of components at runtime to achieve desired functionality. An example of the use of pluggability is the implementation of database independence. You can define a component interface, and as long as you use a component that meets the interface definition, you can plug a variety of database engines into your application.

In .NET, you get pluggability through components.

IDE Compatibility

While perhaps not part of the theoretical definition of a component, all successful implementations of component technology can be used in an IDE. Developers can drag-and-drop components into their application, configure their component, and often see a design-time representation of their component.

In .NET, IDE support is different for components and classes. Components have interactions with the Design window that pure classes do not.

Packaging and Distribution

Components often are made up of classes, icons, bitmaps, and so on. A component technology must have a comprehensive infrastructure for packaging a completed component and distributing it to other developers, who will write code to use the component, as well as end users, who will use applications that use the component.

One of the problems that Microsoft has solved with .NET is that of deploying and using multiple versions of the same component on a given machine.

In .NET, the way that you package and distribute a class is to make a component from it.

Licensing

For a component technology to be successful, there needs to be some mechanism for licensing. Sometimes, developers of a component want to license their component to other developers, but then allow the end users of an application to use the component freely. Licensing is a technology to control intellectual property.

In .NET, you cannot control licensing of a class as you can with a component.

Distributability

One of the major advantages of using components is that you can decide whether a component will run in-process, in a separate process on the same machine, or on a remote machine. The ease with which you can change the distribution of components is a key element of building scalable solutions. Distributed computing is one of the fundamental advantages to building applications in a managed environment such as .NET.

When you are building a distributed system, you will want to write code that creates remote objects. Marshaling is the process of sending objects across boundaries.

You can build a remotable component that is marshaled either by reference or by value. When an object is marshaled by reference, a proxy accesses the object using remote calls. When an object is marshaled by value, a serialized copy of the object is sent to the destination.

Without making a component from a class, you cannot make the decision as to whether the class will run in-process, out-of-process, or on a remote machine.

Security

There needs to be some method to verify that components have the right to do what they are doing. This is only possible in a managed environment with a runtime.

If you want to control certain aspects of security, you need to use components. The security system of the Common Language Runtime (CLR) applies Code Access Security permissions at the assembly level. This means that you can control security for an assembly, but you cannot control security for individual classes within an assembly.

What is a Custom Control?

Custom controls are components that have a visual aspect to them, will be part of the user interface, and will be configured in the Design window. In effect, custom controls are components with additional responsibilities. All custom controls are components, but the reverse is not true.

You can create three basic types of custom controls: one built from scratch, one derived from an existing control, and one that is a composite of existing controls.

Controls Built from Scratch

You can build a custom control from scratch. In practice, this normally means deriving the control from `System.Windows.Forms.Control`.

Building a custom control from scratch is the most flexible approach, but requires the most work. When you build a control from scratch, you are responsible for taking care of the drawing of the control. In addition, you need to handle certain events, such as mouse and keyboard events. You may also want to send events to the user of your custom control.

Controls Derived from an Existing Control

You can build a custom control by deriving from an existing Windows Forms control; for example, you might derive a specialized `TreeView` control from `System.Windows.Forms.TreeView`.

When you create a custom control by deriving from an existing control, you don't normally need to take care of any drawing, because the base class handles that. The base class will also handle most events, including keyboard and mouse events. The base class for this type of custom control is the existing control class.

You would take this approach when there is an existing Windows Forms control that contains most of the functionality that you want. In your custom control, you can handle events from the base class, add functionality that is perhaps driven by properties that you have defined in your custom control, and raise new events for the user of your control.

Controls Built as a Composite of Controls: User Controls

You can build a custom control as a composite of two or more Windows Forms controls. Such a control is termed a *user control* in .NET, and it is normally derived from

For user controls, you build a custom control out of two or more existing controls, called *constituent* controls. You can position the constituent controls relative to each other. Then, when you place your composite control on a form, the constituent controls are placed relative to the position of the composite control. You can handle any of the events of the constituent controls, implement additional functionality that is perhaps driven by properties that you added to your custom control, and raise events.

With this type of custom control, you can build higher-level abstractions. This approach has two advantages:

- Higher-level abstractions reduce the work that the developer using the control needs to do.
- By building composite controls that will be used throughout an application or a set of applications, you can enforce a level of consistency.

 Previous

Next 

 Previous

Next 

Building Windows Forms Custom Controls

We'll start by building a very simple control. But first, let's review the basic steps involved:

- **Generate a DLL, not an EXE.** When you start a new custom control project, the first thing that you need to do is to create a project where the Output Type is a Class Library, not a Windows Application. When you compile, the compiler will generate a DLL, not an EXE. The normal way to do this is to specify the use of the Windows Control Library template in the New Project dialog box. By default, the Windows Control Library template sets the base class to `UserControl`. If you want to build a different type of control, you'll need to change the base class.
- **Write properties.** You need to code any public properties. Properties are an important part of the programming interface to components and controls. You need to carefully consider their design.
- **Code methods.** Write any public (and private) methods.
- **Code event handlers.** Handle any events of the base class. If you are building a composite custom control, you need to handle any necessary events of the constituent controls. Normally, you can use Visual Studio .NET to create the empty event handler and wire up the event handler to the event.
- **Declare and raise events.** If you need to raise any events, you can do so. You can either use existing events or create new events with their own arguments.
- **Implement design-time features.** While implementing design-time features isn't necessary to make a simple custom control, for more elaborate custom controls, this step will make them much easier to use.
- **Customize the Toolbox window.** This step puts your custom control in the Toolbox window so that you can drag-and-drop it onto forms. The control needs to be compiled before you can take this step.

One important note is that if you want your component or custom control to be usable from other .NET programming languages, you must ensure that all public and protected members of the control are Common Language Specification (CLS)-compliant. The C# language is a CLS-compliant language, but there are features in C# (such as unsigned integers) that are not CLS-compliant. If you use those features in the external interface to your control, other developers may not be able to use the control from a different language. You are certainly free to use those features internal to your control, but the external interface should not use them.

Creating a Simple Custom Control

In this first example, we will build a very simple custom control to see how the process works. All this control does is to paint itself yellow and draw *Hello,World* on itself.

1. Start Visual Studio .NET and select File ➤ New ➤ Project. You need to create a new C# Windows Control Library, so set the Template to Windows Control Library. It's also a good idea to check the Create Directory for Solution check box. You will ultimately be adding another project to

the solution to test the control, and it's neater if the solution has a proper directory structure. Name the project `SimpleCustomControl`, and then click the OK button. This will give you a project that will build to a DLL, and some boilerplate code that defines a class for the control. However, don't build the control immediately. There are some changes you need to make to the IDE-generated code first.

2. In the Solution Explorer, right-click `UserControl1.cs` and rename it to `SimpleCustomControl.cs`. After you rename the file, you'll see a dialog box that says, "You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'UserControl1'?" Click Yes. The Design window will look something like [Figure 11-1](#).

Note In Visual Studio .NET 2005, when you rename a file, you are presented with the option of renaming all references to the code element of the same name. This changes the name of the class and the Name property of the custom control. Prior to this version of Visual Studio .NET, renaming the file was only the first step. You also needed to change the class name and the Name property in the Properties window.

3. Change the base class for the control. The default base class, `UserControl`, is specifically intended for controls that act as containers for other standard controls. You will see an example of this in the "[Building Composite Custom Controls](#)" section later in this chapter. In most cases, you will be building custom controls; that is, controls that are derived from `System.Windows.Forms.Control` or some other control class. That will be the case in this example, so change the base class to the `System.Windows.Forms.Control` class:

```
public partial class SimpleCustomControl : System.Windows.Forms.Control {
```

After changing the base class to the `Control` class, the Design window will look something like [Figure 11-2](#). Notice that there is no grid onto which you can drag other controls. Visual Studio .NET supplies the grid only if the control is derived from `UserControl`, for the obvious reason that `UserControl` is the class designed for laying out other contained controls. However, even though you don't have a grid, the Design window is still useful, since it allows access to the Properties window to change events and properties of the control.

Note Sometimes, even though you can see the Design window, it doesn't have the focus. You'll need to click it to give it the focus, so that the Properties window will be populated with the custom control's information.

4. Next, add a Paint event. Press F4 to open the Properties window, and then click the Events button (the yellow lightning bolt) in the toolbar. You'll see the events of the base class for which you can write event handlers. Using the Events pane of the Properties window saves you the effort of writing the empty event handler and wiring the event handler to the event. Double-click the Paint event to create an event handler for it, and modify it as follows:

```
private void SimpleCustomControl_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Yellow, ClientRectangle);
    g.DrawString("Hello, world", Font, Brushes.Black, 0, 0);
}
```

The `ClientRectangle` reference in the call to the `FillRectangle()` method is a property of the `Control` class, so it returns the client rectangle of your custom control, not the `ClientRectangle` of the form that contains your custom control.

5. Build the control now by selecting **Build ➤ Build Solution**.

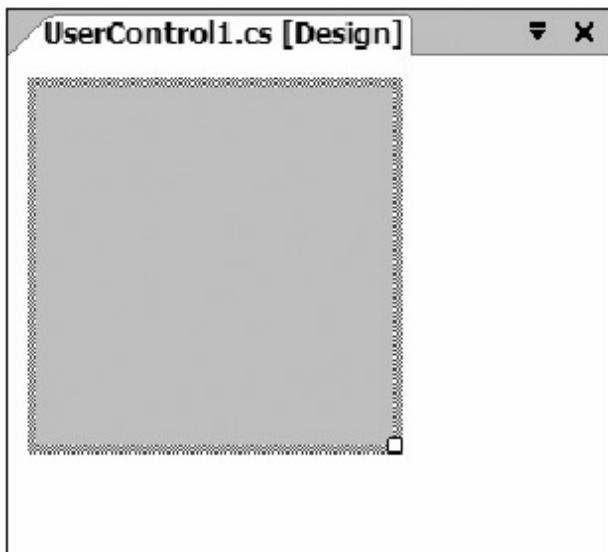


Figure 11-1: Design window for a user control

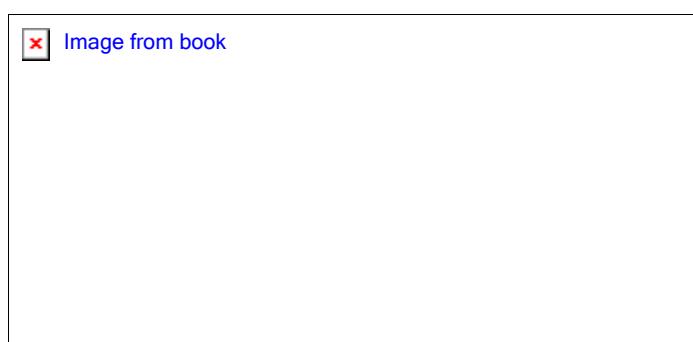


Figure 11-2: Design window after changing the base class

Although you've built the control, if you try to run it, you'll just get an error message complaining that you cannot run a DLL because it doesn't have an entry point. To run it, you will need to create an EXE project to host the control. A custom control is not a complete application. You cannot run it in a stand-alone fashion, but you can write a small test application to test your custom control.

Testing the Simple Custom Control

The general procedure to test a custom control is to compile the control first. Then you create a Windows Forms project in the same solution, and set this project to be the startup project. Then drag your control onto the new form and write whatever testing code is appropriate in the new project.

Allowing multiple projects within one solution is convenient while you're developing a control. It allows you to develop your custom control and your program that tests the custom control in the same instance of Visual Studio .NET. If you modify your custom control or modify your test application, Visual Studio .NET will automatically compile everything that needs to be compiled before running the application. If you were to put the custom control in a different solution, you would need to manually switch to another instance of Visual Studio .NET, compile the custom control, switch back to the Visual Studio .NET for your test application, compile the application, and then run it. Later, after your custom control is fully developed, you can use it in applications without adding the entire project to the solution; you can simply add a reference to it.

Here are the steps for testing the SimpleCustomControl control:

1. Select File ➤ Add Project ➤ New Project and choose to create a new C# Windows Application. Name it `TestSimpleCustomControl`, and then click OK.
2. In the Solution Explorer window, right-click the `TestSimpleCustomControl` project and select Set As StartUp Project. The Solution Explorer will now display the project name in bold, indicating

3. Make sure that the Design window for the form in your test application has the focus (click it). Open the Toolbox window, and you will see the SimpleCustomControl entry. Double-click it, and you will see it placed on your form. If you want, you can move it around on the form or change its size.
4. You can run the application now, and you will see your custom control in the window, as shown in [Figure 11-3](#). It doesn't do much, but there it is. This custom control is about as simple as can be.

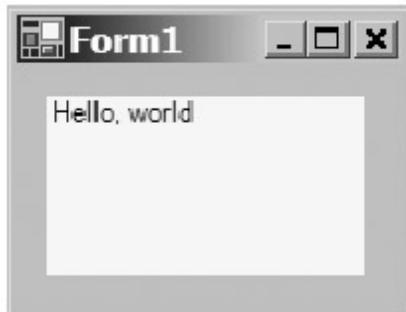


Figure 11-3: The simplest custom control

Customizing the Toolbox

One nice feature of Visual Studio .NET is that, if you are working on other projects in the same solution as the control, the IDE will automatically place your newly built control in the Toolbox. From there, you can drag it onto the form in the same way as you would place a Button, a TextBox, or any other standard Windows Forms control. [Figure 11-4](#) shows our SimpleCustomControl available for use from the Toolbox.

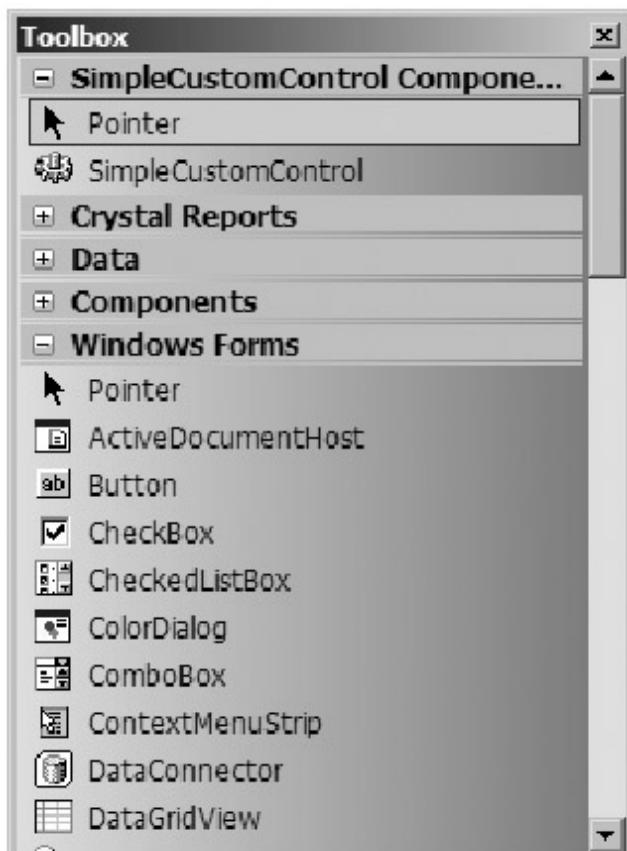


Figure 11-4: Toolbox with the SimpleCustomControl control

However, you may find that the control isn't there. This will probably be the case if you are using the custom control in a production project, rather than simply testing it. That is because, for a production

project, the control project will probably not be part of the same solution as the Windows Forms project. You may also have problems if you built the control before renaming it, because Visual Studio .NET may have put the control in the Toolbox using its old name, and will then be unable to locate it after the name change (that's the reason I told you not to build the control immediately in step 1 of the earlier example in the "[Creating a Simple Custom Control](#)" section). If, for any reason, the control is not already in the Toolbox when you want to add it to another project, you can get it there by customizing the Toolbox.

To add a custom control to the Toolbox, follow these steps:

1. Open the Toolbox, if it is not already open (an easy way to do this is to press Ctrl+W, then X). You must have the Design window open to see the Windows Form tab in the Toolbox.
2. In the Solution Explorer, right-click the control (SimpleCustomControl in this example) and pick View Designer in the context menu.
3. In the Toolbox, right-click the Windows Forms tab and select Choose Items.
4. Click the .NET Framework Components tab, click the Browse button, and navigate to the bin\debug subdirectories in the folder that contains SimpleCustomContol.

Note Obviously, if you are adding a completed and debugged control to the Toolbox, in a production environment, you should navigate to wherever the release version of the control is, not the debug version.

5. You can then see SimpleCustomControl.dll, which you built when you compiled your custom control. Double-click it to add it to the Toolbox. Notice the check box next to the control you just added. By default, this is checked, which means the control now appears in the Toolbox. If you wanted to remove it from the Toolbox, you could simply uncheck this check box.
6. Click OK to close the Choose Toolbox Items dialog box. The control will now be in the Toolbox.

Also note that if your control is built in a different solution, you will need to add a reference to the assembly that defines the custom control in the containing application. You can add this reference by right-clicking the project in the Solution Explorer and selecting Add Reference to bring up the Add Reference dialog box.

By default, when you initially create a project for a custom control, you compile it with debug information in the output file. Before deploying a custom control, you should build a release version. You will also need to implement licensing if required. If you're building a commercial custom control, this step may be necessary.

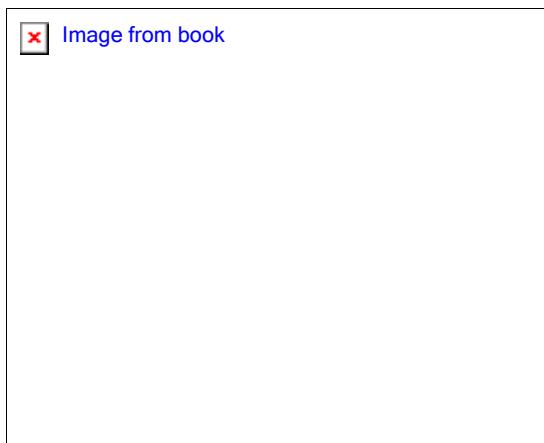
At this point, you have now seen all the steps necessary to build and test a basic custom control. In the following sections, we will develop more sophisticated controls in order to demonstrate some of the principles of coding them.

COMPONENTS VS. CUSTOM CONTROLS

One point worth noting is that in Visual Studio .NET, there is a difference between how you use components and how you use custom controls.

When you use a custom control, you drag it from the Toolbox window onto a grid in the Design window, which you can open for forms and composite custom controls. You will then see a visual representation of your custom control on the grid. You can click the instance of the custom control and bring up the Properties window for it.

In contrast, when you drag a component from the Toolbox window onto a form, the component is visually added to the Component Designer, which is a special area reserved for nonvisible components below the grid in the Design window, as shown in the following example. Working with components is otherwise identical to working with controls. For example, you can click the component to alter its properties.

[Previous](#)[Next](#)[Previous](#)[Next](#)

Adding Properties to Custom Controls

C# and other .NET languages support properties as first-class citizens of languages. There has been some debate as to whether properties should be supported as such. Properties in Java and some other languages are implemented as methods. However, having properties supported as a fundamental part of the language has two main benefits:

- Properties make it easy for reflection to return just the properties of a class. Developers don't rely on special naming techniques (sometimes called a *magic naming convention*). Just as it is bad design to have magic values for variables, I consider magic naming conventions to be bad design. In addition, interactive design environments can more easily allow the programmer using the control to edit its properties.
- When writing code, you can get and set properties as though you were setting and getting member variables of a class. This syntax more accurately reflects the nature of properties.

Creating a Custom Control with Properties

In this section, we will create a custom control that implements a number of properties that determine how the control will display. The control, when running, will look something like [Figure 11-5](#).

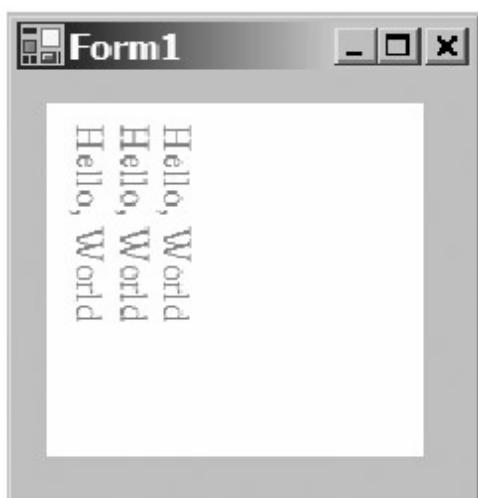


Figure 11-5: Custom control with properties

The control is simply a panel that displays some text a certain number of times either vertically or horizontally. The properties we will add are shown in [Table 11-1](#).

Table 11-1: Properties of the Custom Control

Property	Description
string DisplayText	The text to be displayed
int DisplayCount	The number of times this text will be displayed
Color TextColor	The color the text will be displayed in
Font TextFont	The font used to display the text
TextDirection TextDirection	Whether the text will be displayed horizontally or vertically
Point StartingDisplayPoint	The coordinates within the control for where the first line of text will start

To create the project, create a new C# Windows Control Library project and call it `CustomControlWithProperties`. In the Solution Explorer, right-click `UserControl1.cs` and rename it to `CustomControlWithProperties.cs`. When Visual Studio .NET asks if you would like to rename all references to `UserControl1`, click Yes. Also, as before, use the code editor to change the base class to the `System.Windows.Forms.Control` class.

First, we need an enumeration to describe the text direction, so add the following enumeration to the `CustomControlWithProperties` class. You can add it anywhere within the class, as long as it is not within a method.

```
public enum Orientation
{
    Vertical,
    Horizontal
};
```

Now add the following private member variables to the `CustomControlWithProperties` class. These fields will contain the internal values of the properties listed in [Table 11-1](#).

```
private string displayText;
private int displayCount;
private Color textColor;
private Font textFont;
private Orientation textOrientation;
private Point startingDisplayPoint;
```

Let's look at the implementation of the properties. They will be implemented to simply wrap the fields, but with the extra feature that when any of these properties is set, we will need to have the control repainted to show its updated appearance. We can do this by calling the `Control.Invalidate()` method from within the `set` property accessor. For the `DisplayText` property, this means the code will look like this:

```
public string DisplayText
{
    get
    {
        return displayText;
    }
    set
    {
        displayText = value;
        Invalidate();
    }
}
```

The remaining properties are implemented in a similar manner:

Pro NET 2.0 Graphics Programming

```
public int DisplayCount
{
    get
    {
        return displayCount;
    }
    set
    {
        displayCount = value;
        Invalidate();
    }
}
```

```
public Color TextColor
{
    get
    {
        return textColor;
    }
    set
    {
        textColor = value;
        Invalidate();
    }
}
```

```
public Font TextFont
{
    get
    {
        return textView;
    }
    set
    {
        textView = value;
        Invalidate();
    }
}
```

```
public Orientation TextOrientation
{
    get
    {
        return textOrientation;
    }
    set
    {
        textOrientation = value;
        Invalidate();
    }
}
```

```
public Point StartingDisplayPoint
{
    get
    {
        return startingDisplayPoint;
    }
    set
    {
        startingDisplayPoint = value;
        Invalidate();
    }
}
```

Note that if changing the property changes the appearance of our custom control, then it is important that we call the `Invalidate()` method. When running in design mode, this also causes Visual Studio .NET to redraw the control when the property changes.

Next, we need to add a `Paint` event handler to our custom control and modify the code as follows:

```
private void CustomControlWithProperties_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    PointF point = StartingDisplayPoint;
    Brush brush = new SolidBrush(TextColor);
    StringFormat sf = new StringFormat();
    if (TextFont == null)
        TextFont = new Font("Times New Roman", 12);
    if (textOrientation == Orientation.Vertical)
        sf.FormatFlags = StringFormatFlags.DirectionVertical;
    for (int count = 0; count < DisplayCount; ++count)
    {
        g.DrawString(DisplayText, TextFont, brush, point.X, point.Y, sf);
        if (textOrientation == Orientation.Vertical)
            point.X += TextFont.GetHeight();
        else
            point.Y += TextFont.GetHeight();
    }
}
```

The code for this handler should be fairly self-explanatory. It causes the text to be painted at the appropriate location using the specified color, text direction, and font. Note the use of the `StringFormat` class, which, as you saw in [Chapter 4](#), is defined in the `System.Drawing` namespace. This class contains information about how to draw a string, and can be passed as a parameter to the `Graphics.DrawString()` method. Here, we use `StringFormat` to indicate if the text should be displayed vertically.

Now we need to build the custom control (press `Ctrl+Shift+B`) and, if necessary, customize the Toolbox so that it contains the control. (See the "[Creating a Simple Custom Control](#)" section earlier in this chapter for detailed instructions for building a custom control.)

Testing the Custom Control with Properties

To create a test application, add a new Windows Application project to the solution and name it `TestCustomControlWithProperties`. Set this to be the startup project by right-clicking it in the Solution Explorer and clicking Set As StartUp Project.

In the Toolbox, double-click the `CustomControlWithProperties` entry. This will add the control to your form. Resize the control so it is substantially larger than the default size that Visual Studio .NET gives it.

At the bottom of the Properties window for this control, locate the entries for all of the properties that we added to this custom control. If you click to order the properties by category, you will find all the new properties together under the Misc category. Set the properties to the values shown in [Table 11-2](#).

Table 11-2: Property Values for the Custom Control

Property	Value
DisplayCount	3
DisplayText	Hello, World
StartingDisplayPoint	6, 6
TextColor	Red
TextFont	Times New Roman, 12 point

TextOrientation

Vertical

You should notice a few things about the way that Visual Studio .NET displays the properties. In all these cases, Visual Studio .NET is able to customize the way you can edit the properties, because it recognizes the data types of the properties.

The StartingDisplayPoint property is a "composite" property, made up of child properties. You can change the value of the point by entering the X and Y values separated by a comma, or you can expand the property by clicking the + to the left of the property and entering the X and Y values separately. This behavior is automatically implemented for the Point data type. In the [next chapter](#), when we examine design-time support in detail, you will see how to implement this for your own composite properties.

The TextOrientation property is implemented with a drop-down list. Visual Studio .NET will do this for enums by default.

Both the TextColor and the TextFont properties have a custom UI editor. This means that you can click a button in the Properties window and edit the property in a more elaborate dialog box. This can either be a drop-down window or a modal dialog box displayed in the center of the screen. The TextColor property offers a drop-down window. The TextFont property's dialog box is modal style. In the [next chapter](#), you will see how to implement your own custom UI editors for the Properties window.

Previous

Next

Previous

Next

Building Custom Controls with Default Property Values

In the CustomControlWithProperties example, all the properties were initialized with the usual default values for variables—a blank string, zero, a null font, or a null color. In most cases, it is better to have properties initialized to some meaningful values by default. This makes your control easy for other developers to use.

There are two aspects to using default property values. You need to make sure that your properties are initialized in the code for the control, and also that Visual Studio .NET understands the default values. The latter point is important because Visual Studio .NET, when aware of default values, is able to take several steps to make editing the properties easier:

- If the value of the property is not the default value, Visual Studio .NET will display the value of the property in bold. For example, the default value for the Text property for a form or control is the empty string. If you set this property to anything else, it will be displayed in bold.
- Visual Studio .NET will provide a context menu for the property that allows the programmer using the control to reset the value to the default value. The programmer is able to right-click the property value and pick Reset Value from the context menu.
- Visual Studio .NET can generate slightly more efficient code.

You can see this behavior for the properties of the forms and controls that are part of the Windows Forms framework.

You must take two steps to initialize your properties to default values and have Visual Studio .NET process these values properly: initialize your member variables and set an attribute on the property. You'll see how this works in the next example.

Creating a Custom Control with Default Property Values

We will develop the CustomControlWithProperties control we built in the [previous section](#) so that it supports default properties. First re-create the CustomControlWithProperties control, except this time give it (and the class that represents the control) the name CustomControlDefaultPropValues. Now we can make the necessary changes to it.

When you declare the private member variables that contain your property values, you initialize them to the default value. Change the member variable declarations in our CustomControlDefaultPropValues class as follows:

```
private string displayText = "Hello, World";
private int displayCount = 3;
private Color textColor = Color.Red;
private Font textFont = new Font("Times New Roman", 12);
private Orientation textOrientation = Orientation.Vertical;
private Point startingDisplayPoint = new Point(6, 6);
```

Next, we need to use a mechanism to inform Visual Studio .NET of our default value. You can use two mechanisms to do this.

One way to tell Visual Studio .NET about a property's default value is to set an attribute before the declaration of the property. Add the following attributes to the `DisplayText`, `DisplayCount`, and `TextDirection` properties:

```
[DefaultValue("Hello, World")]
public string DisplayText
{
    get
    {
        return displayText;
    }
    set
    {
        displayText = value;
        Invalidate();
    }
}

[DefaultValue(3)]
public int DisplayCount
{
    get
    {
        return displayCount;
    }
    set
    {
        displayCount = value;
        Invalidate();
    }
}

[DefaultValue(Orientation.Vertical)]
public Orientation TextOrientation
{
    get
    {
        return textOrientation;
    }
    set
    {
        textOrientation = value;
        Invalidate();
    }
}
```

This approach is fine in the cases where the properties are of types whose values can be listed as a parameter in an attribute: a string, a number, or an enumeration. However, in the cases where the properties return some other type, you need to use another approach. `DefaultValue` won't work, for example, with `Color`, because there is simply no way in .NET to specify a color as a parameter attribute. There isn't any syntax when specifying attributes to specify that the attribute should run a constructor, so when a property is contained in a structure or object, you must write some code.

For cases where the properties return some type other than a string, a number, or an enumeration, you use a technique based on the `Reset<PropertyName>` and `ShouldSerialize <PropertyName>` methods. With this technique, you supply methods to the class that can reset the property to its default

value and to compare a given property to the default value. Visual Studio .NET is able to recognize these methods from their names. More specifically, you define a method, `Reset<PropertyName>()`, which resets the property to its default value, and a method, `ShouldSerialize<PropertyName>()`, which checks whether the property has its default value. For the `TextColor` property, these methods look like this:

```
public void ResetTextColor()
{
    TextColor = Color.Red;
}

public bool ShouldSerializeTextColor()
{
    return TextColor != Color.Red;
}
```

For the remaining properties, the `Reset...()` and `ShouldSerialize...()` methods are as follows:

```
public void ResetTextFont()
{
    TextFont = new Font("Times New Roman", 12);
}

public bool ShouldSerializeTextFont()
{
    return ! TextFont.Equals(new Font("Times New Roman", 12));
}

public void ResetStartingDisplayPoint()
{
    StartingDisplayPoint = new Point(6, 6);
}

public bool ShouldSerializeStartingDisplayPoint()
{
    return StartingDisplayPoint != new Point(6, 6);
}
```

Add these methods to the `CustomControlDefaultPropValues` class and compile the custom control.

Note If you are being very observant, you may have noticed that this technique is reminiscent of the old magic naming techniques that are normally regarded as bad programming practice these days. This is arguably an unfortunate design, but in this case, it is the way Visual Studio .NET has been designed, so we will have to live with it.

Testing the Custom Control with Default Property Values

Now we are ready to add a test application. As before, create a Windows Forms application and use the Toolbox to add the custom control to the form. Immediately after doing this, you will see that the values are not displayed in a bold font, because the current values are the default values. Change some values, and Visual Studio .NET will display the property values in bold, indicating that those values are no longer the defaults. Right-click a changed value so that you can see the context menu that allows you to reset the property value.

Of course, this doesn't change the runtime behavior of the control. However, it does make it a bit more convenient for developers to use the control when writing code with Visual Studio .NET.

Note For each property, you should either use the `DefaultValue` attribute or the `Reset<PropertyName>()` and `ShouldSerialize<PropertyName>()` methods, but not both.

Examining the Code Generated by Changed Property Values

I said earlier that informing Visual Studio .NET of the default property values allows it to generate slightly more efficient code. In this section, you will see how this works by examining the code generated when

In the CustomControlDefaultPropValues sample, change some of the properties to values that are not their defaults. Change the property values as shown in [Table 11-3](#).

Table 11-3: New Property Values for the Custom Control

Property	Value
DisplayCount	5
DisplayText	Goodbye, World
TextColor	Lime

Now open the code editor for the form that contains the custom control to see what code the IDE has added to implement these new values. Click the Show All Files button in the toolbar at the top of the Solution Explorer window. Then click the + to the left of `Form1.cs`. This allows you to see the `Form1.Designer.cs` file, which contains the code that Visual Studio generates for the use of this custom control. Open `Form1.Designer.cs`. In the code editor, expand the code by clicking the + symbol to the left of the Windows Form Designer-generated code.

Find the code that initializes the properties to their default values. It should look something like this:

```
//  
// customControlDefaultPropValues1  
//  
this.customControlDefaultPropValues1.DisplayCount = 5;  
this.customControlDefaultPropValues1.DisplayText = "Goodbye, World";  
.  
.  
.  
.  
.  
.  
this.customControlDefaultPropValues1.TextColor = System.Drawing.Color.Lime;  
.  
.
```

You can see the three lines of code that are responsible for setting the properties, which you changed from their default values. Since the other two properties are set to their default values, Visual Studio .NET knows not to generate the code to set them. They will be set to their default values when the custom control is constructed. That is where the savings come in: if Visual Studio .NET didn't know that these were the default values, it would supply redundant code to initialize the values again. For a simple control such as this one, it is a little more efficient, but for a very complex control that contained a number of constituent controls, the savings could become significant.

Note that it is important that you initialize the member variables to the same values that you indicate in either the `DefaultValue` attribute or the `Reset<PropertyName>()` and `ShouldSerialize<PropertyName>()` methods. This is a manual process, with no automated way to ensure that the custom control developer does it correctly. If you do it incorrectly, the programmer using the custom control would encounter some confusing behavior.

 Previous

Next 

 Previous

Next 

Building Focusable Controls

In this section, we will explore another aspect of making sure your controls behave correctly. We will look at what happens when the control gains or loses the focus.

You will want most of your custom controls to be able to participate in the focus mechanism of Windows Forms. This involves the following:

- Allow the users to tab to the control, as well as off it to other controls.
- Give the users some visual indication that the control has focus when they tab to the control.
- If desired, allow the focus to be transferred to the control via a mnemonic (a hotkey). If the focus is not currently held by another control that can take text input (such as a `TextBox`), the user can press

a hotkey (or Alt plus that key) to move the focus to the control and, in some cases, cause the control to activate (for example, a CheckBox control will check or uncheck itself).

- If the control is disabled, have its appearance reflect that.

There is one concept that is important to a discussion of the implementation of focusable controls: *focus cues*.

Understanding Focus Cues

One aspect of the Windows user interface that many developers might not be aware of is that many controls, including the Button control, have two separate indications of focus. Consider a button that does not have the focus:



Now look at a Windows button with focus. Note the subtle difference here—a black outline around the button.



Finally, here's a Windows button with focus, but this time there is also a dotted line around the text, and the mnemonic character (hotkey), O in this case, is underlined:



The difference between the two OK buttons with focus is that for the first one, the user has only been using the mouse in the containing control. The second shows the situation if the user has also been using the keyboard to navigate between controls (for example, by pressing the Tab key).

In more technical terms, in the second OK button with focus example, the *keyboard focus cues* have been activated. In Windows 2000 and later, a behavior hides keyboard focus cues until the user performs a keyboard action such as tabbing. The idea here is that if the user operates the user interface only with the mouse, by hiding keyboard focus cues, the user interface is cleaner looking and has less visual noise. Note that the dotted line appears when the user tabs between controls, but the underlining of the hotkey occurs when the user presses the Alt key.

To determine if the control has focus, you use the `ContainsFocus` property. To determine if you should show keyboard focus cues, you use the `ShowFocusCues` property. To have your controls behave properly, your drawing routines should reflect this usage of these two properties.

To see this behavior, you need to ensure that Windows 2000 or Windows XP is configured correctly. You can find the appropriate setting in the Display Properties dialog box (opened through the Control Panel or by right-clicking the desktop), under the Effects tab. In Windows 2000, make sure that the Hide Keyboard Navigation Indicators Until I Use the Alt Key check box is checked. In Windows XP, in the Display Properties dialog box, under the Appearance tab, click the Effects button. In the Effects dialog box, make sure that the Hide Underlined Letters for Keyboard Navigation Until I Press the Alt Key check box is checked.

Implementing Focusable Controls

The general steps for constructing a simple custom control that implements focusing functionality correctly are as follows:

- Write a generalized drawing routine that shows focus cues (that is, those focus cues that are always present, whether or not the user uses the keyboard) if appropriate, which also shows keyboard focus cues, if appropriate, and draws the control correctly based on whether it is enabled or disabled.

- Write a `Paint` event that calls the drawing routine.
- Write an event handler for the `ChangeUICues` event. This is the event that is raised if Windows detects that some aspect of the focus or the focus cues may have changed.
- Repaint the control when you get the `GotFocus` event that indicates the control has just taken the focus.
- Repaint the control when you get the `LostFocus` event that is raised when the control has lost the focus.
- Override the `ProcessMnemonic()` method. This method is defined in the `Control` class. It should be used to check if a key the user has pressed is the mnemonic key for that control, and if so, take the appropriate action. The default implementation in `Control` does nothing.
- Handle the `Click` event, or any desired mouse events.
- Write any desired `KeyDown` or `KeyUp` event handlers so that the user can operate the control using the keyboard.

You'll go through these activities to build a simple focusable custom control in the [next section](#).

Creating a Focusable Control

In this section, we will build a control that exhibits correct behavior for focusing. The control will act rather like a button, except that instead of using the `Button` class, you will code all the drawing and responding to user events yourself, just so you can see how it can be done.

Since we don't want to get bogged down in code to draw the raised or sunken appearance of buttons, the control will just look like a panel, but when it has the focus, it will display with a selected background color, `Color.LightBlue` by default (although we will build the ability to change this color programmatically into the control). [Figure 11-6](#) shows the control.



Figure 11-6: Focusable control

In [Figure 11-6](#), the text of the control has been set to the string `&Focus`, as will be done in the example. This text is built into the control, but can be set when an instance of the control is created. The `&` is not itself displayed, but indicates that the following character, `F`, is to be the hotkey for this instance of the control.

If the keyboard cues are to be displayed, the control will be bordered with a dotted line, just like a real button (and of course the hotkey will be underlined), as shown in [Figure 11-7](#).



Figure 11-7: FocusableControl with focus

If the control does not have the focus, it won't display its background in light blue, but will simply use the default control background color.

Clicking the button will cause a dialog box to pop up, saying that the control was clicked, as shown in [Figure 11-8](#).



Figure 11-8: Clicking the focusable button

If you press the Enter key or spacebar while the control has the focus, you will get a similar dialog box, saying that one of those keys has been pressed.

The control can also display itself grayed if it is disabled, as shown in [Figure 11-9](#).



Figure 11-9: Disabled focusable control

Creating the Control

As usual, the first step is to create a new Windows Control Library project, this time calling it FocusableControl. Change the UserControl1.cs file to FocusableControl.cs and the base class to System.Windows.Forms.Control.

At the top of the FocusableControl.cs module, add using statements for both the System.Drawing.Drawing2D namespace and the System.Drawing.Text namespace, since we will be using classes from these namespaces:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Data;
using System.Windows.Forms;
```

Add the following private member variable to the FocusableControl class:

```
private Color focusColor = Color.LightBlue;
```

This field will contain the color to be used as the background color when the control has the focus. We will allow this color to be changed, and clearly, if it is changed, the control will need to be redrawn, so we add the following public property:

```
public Color FocusColor
{
    get
    {
        return focusColor;
    }
    set
    {
        focusColor = value;
    }
}
```

We also want the `FocusColor` property to display correctly in the Properties window: bold if it doesn't have its default value. We will use the `Reset/ShouldSerialize` technique this time, as demonstrated in the previous example. Add the following code:

```
public void ResetFocusColor()
{
    FocusColor = Color.LightBlue;
}

public bool ShouldSerializeFocusColor()
{
    return FocusColor != Color.LightBlue;
}
```

Painting the Control

Let's examine the code we will need to paint the control. When writing a custom control such as this, there are a number of circumstances where you need to redraw your control based on a changed state. As an example, if you have a property that changes the color of your custom control, changing the property will require that the control be redrawn. The most effective way to do this is to write one routine that draws your control based on the current state, and have this routine called whenever something has happened that would mean the appearance of the control may have changed or when the `Paint` event has been raised.

Add the following private method to the class:

```
private void drawButton(Graphics g)
{
    if (ContainsFocus)
    {
        // Draw in focus color

        Brush b = new SolidBrush(FocusColor);
        g.FillRectangle(b, ClientRectangle);
        b.Dispose();
    }
    else
    {
        // Draw in BackColor
        Brush b = new SolidBrush(BackColor);
        g.FillRectangle(b, ClientRectangle);
        b.Dispose();
    }

    // The StringFormat class centers the text in our custom
    // control. In addition, it handles the correct drawing of
    // the mnemonic character.
    StringFormat sf = new StringFormat();
    sf.Alignment = StringAlignment.Center;
    sf.LineAlignment = StringAlignment.Center;
    if (ShowKeyboardCues)
        sf.HotkeyPrefix = HotkeyPrefix.Show;
    else
        sf.HotkeyPrefix = HotkeyPrefix.Hide;
    if (Enabled)
    {
        // If the control is enabled, the color of the text is ForeColor
        Brush fb = new SolidBrush(ForeColor);
        g.DrawString(Text, Font, fb, ClientRectangle, sf);
        fb.Dispose();
    }
    else
    {
```

```
// If the control is not enabled, first draw the text in white, offset
// one pixel up and to the left, and then draw the text in DarkGray
g.TranslateTransform(-1, -1);
g.DrawString(Text, Font, Brushes.White, ClientRectangle, sf);
g.ResetTransform();
g.DrawString(Text, Font, Brushes.DarkGray, ClientRectangle, sf);

}

// Draw a dotted line inside the client rectangle
if (ShowFocusCues && ContainsFocus)
{
    Rectangle r = ClientRectangle;
    r.Inflate(-4, -4);
    r.Width--;
    r.Height--;
    Pen p = new Pen(Color.Black, 1);
    p.DashStyle = DashStyle.Dot;

    g.DrawRectangle(p, r);
}
}
```

Quite a lot is going on in this method. We start off by testing to see if the control has the focus, and if so, shade in the background color:

```
if (ContainsFocus)
{
    .
    .
    .
}
```

`ContainsFocus` is a Boolean property that our class inherits from the `Control` class. It returns true if the control has the focus.

The next part of the code has to do with displaying the text for the control. For this, we use that helper class, `System.Drawing.StringFormat`, again. We set the `LineAlignment` properties of the `StringFormat` object to instruct `DrawString()` to center the text horizontally and vertically. More interesting is the code to control underlining of the hotkey:

```
if (ShowKeyboardCues)
    sf.HotkeyPrefix = HotkeyPrefix.Show;
else
    sf.HotkeyPrefix = HotkeyPrefix.Hide;
```

`ShowKeyboardCues` is another Boolean property that is implemented in the `Control` class. This one returns true if the system detects that we should be showing keyboard cues—in other words, if the user has started using keys to navigate around the form. If keyboard cues are to be shown, we will want to underline the hotkey (or mnemonic key) for this control, and we can get `Graphics.DrawString()` to do this by setting the `StringFormat.HotkeyPrefix` property to the enumerated value `HotkeyPrefix.Show`. (The `HotkeyPrefix` enumeration is defined in the `System.Drawing.Text` namespace.) `Graphics.DrawString()` will identify the hotkey as the first character that is preceded by an & in the text it is asked to print.

Having set up the `StringFormat` object, we then draw the string, in the normal foreground color if the control is enabled, or grayed if it is disabled.

Finally, if keyboard cues are enabled, we need to draw the dotted line around the control. That is the purpose of the final if block in the `drawButton()` method:

```
// Draw a dotted line inside the client rectangle
if (ShowFocusCues && ContainsFocus)
{
    .
    .
    .
}
```

If the control has the focus when this routine draws the background color of the custom control, it draws

the control in the `FocusColor`, regardless of whether we should show the keyboard focus cues.

However, we draw the dotted line inside the client rectangle only if both `ShowFocusCues` and `ContainsFocus` are true.

Invoking the Method to Paint the Control

Now that we have defined the `drawButton()` method, we need to ensure it is called at appropriate times. The most obvious case is when the `Paint` event is raised. So, we will add a `Paint` event handler. Do this in the usual way: by clicking the Events button in the Properties window of the Designer view, and then double-clicking the `Paint` event. The code we need to add to our handler is as follows:

```
private void FocusableControl_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    drawButton(e.Graphics);
}
```

We also need to add handlers for those times when the appearance of the control changes. This can happen in the following cases:

- The user interface cues change.
- The control gains or loses focus.
- The control is enabled or disabled.
- The mnemonic key is pressed. (We will deal with this case in the [next section](#).)

These are the cases that we must deal with in our code. There are other cases, such as when the control is shown or hidden, but those are already taken care of by the `Control` class.

First, let's cover the case of when user interface cues change. At various times, Windows Forms will determine that the user interface cues should be changed. As an example, if keyboard focus clues are not currently being shown, and the user presses a key, Windows Forms may decide that keyboard cues should now be shown. When this happens, Windows will raise the `ChangeUICues` event. We implement an event handler for this as follows:

```
private void FocusableControl_ChangeUICues(
    object sender, System.Windows.Forms.UICuesEventArgs e)
{
    if (e.ChangeFocus || e.ChangeKeyboard)
    {
        Graphics g = CreateGraphics();
        drawButton(g);
        g.Dispose();
    }
}
```

Note that this example implements the handler by calling `drawButton()` directly. If you prefer, you could simply call `Control.Invalidate()` to get a `Paint` event raised, which would have the same effect.

Now let's consider the cases when the control gains or loses the focus. For simple cases, the `Control` class provides `Enter` and `Leave` events. However, we won't use those. Instead, we will provide event handlers for the `GotFocus` and `LostFocus` events. These events are similar to `Enter` and `Leave`, but work at a lower level. Using them is recommended when you are working with tasks like controlling the UI cue-painting event manually, as we are doing here.

Now, ordinarily, when you implement an event for a custom control, you can have Visual Studio .NET do most of the housekeeping. You can open the Design window for your custom control, and then open the Properties window. You can click the Events button (the small yellow lightning bolt) to see the list of events. You can then double-click the event that you need to implement, and the IDE will automatically create an empty event handler, and also automatically register your event handler in the `InitializeComponent()` method, which is in the Component Designer-generated Code section in the code editor for your control.

However, working with the `GotFocus` and `LostFocus` events is different. These events have the `Browsable` attribute set to false, so you cannot see these events in the Properties window, reflecting

the fact that these are relatively low-level events that you should be providing handlers for only if you are doing fairly advanced stuff. Indeed, if you don't take care with the code that handles these events, you can cause your application to stop responding. In these two events, you are supposed to do one thing, and one thing only, which is to modify the appearance of your custom control.

Since the `Browsable` property is set to `false`, we need to register our event handler manually, which means modifying the code in the constructor to the control.

```
public FocusableControl()
{
    InitializeComponent();

    this.GotFocus += new EventHandler(FocusableControl_GotFocus);
    this.LostFocus += new EventHandler(FocusableControl_LostFocus);
}
```

Now we can implement the event handlers:

```
private void FocusableControl_GotFocus(
    object sender, System.EventArgs e)
{
    Graphics g = CreateGraphics();
    drawButton(g);
    g.Dispose();
}

private void FocusableControl_LostFocus(
    object sender, System.EventArgs e)
{
    Graphics g = CreateGraphics();
    drawButton(g);
    g.Dispose();
}
```

Enabling the Mnemonic Key

When the user presses a key, then Windows Forms gives each control the chance to check if that key is the mnemonic for that control and, if so, to process the keystroke. Windows does this by calling the virtual method, `Control.ProcessMnemonic()`, against each control, passing it as a parameter the character that has been pressed. If all controls decline to process it, then the event bubbles up to container controls and forms, until a control indicates that it has accepted the keypress by returning `true` from `ProcessMnemonic()`. The default `Control` implementation of `ProcessMnemonic()` does nothing and always returns `false`.

To process the mnemonic keystroke, you need to override the `ProcessMnemonic()` method of your control. If your control is both enabled and visible, and if the character that was pressed is the mnemonic character for your control, then your method should take the appropriate action and return `true`. Otherwise, your method should return `false`.

Add the `ProcessMnemonic()` method to our class, as follows:

```
protected override bool ProcessMnemonic(char charCode)
{
    if (Enabled && Visible && IsMnemonic(charCode, this.Text))
    {
        // Perform action associated with mnemonic
        // Moves focus to our control
        Focus();
        Graphics g = this.CreateGraphics();
        drawButton(g);
        g.Dispose();
        MessageBox.Show("Mnemonic pressed");
        return true;
    }
    return false;
}
```

One item of note is the use of the `IsMnemonic()` method. This is a static method defined in `Control`, and it compares the character and string passed in to see if the character is the hotkey for that string (that is to say, the first character that is preceded by an &). If it is, then `IsMnemonic()` returns true; if not, it returns false. We can pass the character code and the text of our control to this method. This does depend on setting the text of the control to something that identifies a hotkey, but we will do so in our code, since we will give our control the text &Focus. The `IsMnemonic()` method, along with the special mnemonic functionality in the `StringFormat` class, makes it easy to implement mnemonic functionality.

Note that because the system will automatically call `ProcessMnemonic()` against the controls on the form when the user presses a key, we do not need to explicitly write any code to invoke the method.

Responding to User Events

We won't implement full button-style functionality with mouse events (in other words, pressing, releasing, trapping the mouse, and so on), but we will add an event handler for the `Click` event. The `Click` event is browsable, so you can add the event handler in the automated way. Here is the `Click` event handler:

```
private void FocusableControl_Click(
    object sender, System.EventArgs e)
{
    Focus();
    MessageBox.Show("Control clicked");
}
```

The first thing that this method does when it gets this event is to move the focus to our control. It then takes whatever action it should take when the control is clicked. For now, the method puts up a dialog box.

Let's also add an event handler for keyboard events, so that when the control has focus, the user can press the spacebar or the Enter key. There are three keyboard events, each for a different purpose, as described in [Table 11-4](#).

Table 11-4: Keyboard Events

Event	Purpose
KeyDown	Occurs once when the key is pressed. This event is used to operate controls that are not used for editing text, such as buttons, tree controls, and so on.
KeyPress	Occurs when the key is pressed. If the user holds down the key, the event repeats at the rate specified in the Keyboard Properties dialog box, which is opened from the Control Panel. This event is used in a control that edits text.
KeyUp	Occurs once when the key is released. This event can be used in conjunction with the <code>KeyDown</code> event to give some type of visual feedback when the user presses and releases keys. For example, if you tab to a button in a dialog box and press the spacebar, the button is visually pressed. When you release the spacebar, the button is visually released.

For the purposes of this control, we want to implement a handler for the `KeyDown` event:

```
private void FocusableControl_KeyDown(
    object sender, System.Windows.Forms.KeyEventArgs e)
{
    if (e.Alt == false &&
        e.Control == false &&
        e.Shift == false &&
        (e.KeyCode == Keys.Space || 
        e.KeyCode == Keys.Enter))
    {
        MessageBox.Show("Space or Enter pressed");
        e.Handled = true;
        return;
    }
}
```

In this event handler, we need to make sure that the Alt key, the Ctrl key, or the Shift key is not down. If this is true, and the user pressed either the spacebar or the Enter key, we take the appropriate action.

Testing the Focusable Control

Our FocusableControl control is now ready, so we need to write a container form to test it. To the existing project, add a new Windows Application named TestFocusableControl and set it as the startup project using the usual procedure.

Our test harness will be a Windows Forms application that contains an instance of FocusableControl, along with two real buttons. One of the buttons is used to show or hide the control; the other is used to enable or disable it. When running, the application will look something like [Figure 11-10](#) (when keyboard cues are in use).

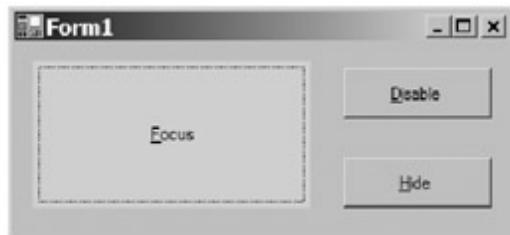


Figure 11-10: Using FocusableControl in a test application

Once you have created the project, you should use the Visual Studio .NET Design view to add a FocusableControl object to the form from the Toolbox. Resize it so it is a suitable size and set the Text property of the control to &Focus. Then, add two Button controls to the form.

Set the first button's Name property to btnEnableDisable and its Text property to &Disable. Set the second button's Name property to btnShowHide and its Text property to &Hide.

Double-click the btnEnableDisable button, and modify the Click event as follows:

```
private void btnEnableDisable_Click(object sender, System.EventArgs e)
{
    btnEnableDisable.Text =
        focusableControl1.Enabled ? "&Enable" : "&Disable" ;
    focusableControl1.Enabled = !focusableControl1.Enabled ;
}
```

Double-click the btnShowHide button, and modify the Click event as follows:

```
private void btnShowHide_Click(object sender, System.EventArgs e)
{
    btnShowHide.Text = focusableControl1.Visible ? "&Show" : "&Hide" ;
    focusableControl1.Visible = !focusableControl1.Visible;
}
```

Now you can compile and run the test application, and see the behavior that we have implemented:

- When you start the test application, you can see that our control has the focus. It has the focus first because it was the first control that you added to the application.
- If you tab, you can see the control lose and gain focus and, since you are using the keyboard to move between controls, you will see the keyboard cues appear on both our control and the real buttons.
- If you disable the control, you can see the disabled appearance.
- While disabled, you also can see that the control no longer participates in the tabbing sequence.
- You can click the control, which both moves the focus to it and shows a message box.
- You can hide or show the control.
- When the control has the focus, you can press the spacebar or Enter key, and you will see a

- If the control is enabled and visible, you can press the F key on the keyboard, and you will see a message box. If the control is disabled or invisible, the mnemonic is no longer active.

Also note that if you start the test application from Visual Studio .NET using the keyboard, it starts with the keyboard focus cues visible. If you want to see the test application without the keyboard focus cues,

start it by using the mouse. From the menu bar, select Debug ➔ Start. The application starts, and the keyboard focus cues are hidden. Now, press the Tab key, and you can see the keyboard focus cues appear. You can see that this custom control has the same focus behavior as the native Windows buttons.

◀ Previous

Next ▶

◀ Previous

Next ▶

Generating Events

As I've mentioned earlier, events form an important part of Windows Forms architecture. In the examples you've seen so far, you have been freely using some of the basic events defined by Microsoft, such as Paint. However, you may also find that you need to define and raise your own events that are relevant to your custom controls.

Events are implemented via delegates, so first, we will quickly review delegates.

Reviewing Delegates

A *delegate* is a reference type that refers to either a static or instance method of a class. The definition of a delegate includes the specification of the signature of a method. This signature must match the signature of the method that is used to declare the delegate.

Delegates are similar to *function pointers*, but have some advantages over them. Function pointers can refer only to a static method, or a method not contained in a class, whereas delegates can refer to instance methods as well. Delegates are type-safe, and work well within object-oriented languages.

When you instantiate a delegate, you can instantiate it with a reference to both an object and a method. The delegate will then execute in the context of that object. The code in the method referred to by the delegate can refer to the `this` object, and the `this` object is the object with which you instantiated the delegate.

Another important difference between pointers to functions and delegates is that you can compose delegates of more than one method. This is called a *multicast delegate*. When you call the delegate, it will invoke more than one method, which means that you can have more than one event handler for a given event. This is sometimes useful.

It is invalid to declare a multicast delegate that returns a value. If each method in a delegate returns a value, what would be the return value of a call to the delegate?

To define, declare, and use a delegate is a three-step process:

- Write a delegate definition.
- Declare and instantiate an object of the type of our delegate definition.
- Invoke the method(s) to which the delegate refers.

A delegate definition is as follows:

```
public delegate void MyDelegate(Object obj, EventArgs e);
```

You can declare a delegate as a member variable of your class, like this:

```
public MyDelegate MyEvent;
```

If you have an event handler declared like this:

```
private void MyEventHandler(Object obj, EventArgs e)
{
```

```

/*
code to handle the event goes in this method.
*/
Console.WriteLine("in MyEventHandler");
}

```

then you can initialize your delegate variable like this:

```
this.MyEvent += new MyDelegate(this.MyEventHandler);
```

Having declared your delegate, you could now invoke it:

```
Object obj = this; // This code is operating in the context of an object
                  // that is going to raise the event.
EventArgs args = new EventArgs(); // Initialize args as appropriate
                                  // after constructing it.
MyEvent(obj, args); // Invoke the delegate.
```

An *event* is a special type of a multicast delegate. It is slightly different from a multicast delegate in two ways: there are certain restrictions on the signatures of event handlers, and C# has some additional syntactical constructs to make it easier to use events.

The only difference between declaring an event and a delegate is the use of the `event` keyword in the member variable declaration:

```
public event MyDelegate MyEvent;
```

This use of the `event` keyword enables Visual Studio .NET to put `MyEvent` in the Events pane of the Properties window.

Adding a Custom Event to a Control

Now that we have reviewed delegates and events, let's add an event to our `FocusableControl` example. At present, the event handlers simply display message boxes when the control is clicked or the spacebar, hotkey, or Enter key is pressed. We will modify these so that they raise an event that we have defined. It is then up to the containing form (or any other interested party) to handle this event however it chooses, which, for our example, will once again be by displaying a message box.

We will be adding a substantial amount of code, but it won't change much in terms of what the application does in this example. However, the real significance is the way that linking these actions into the event architecture allows other controls or classes to handle the event if required. Using events also generally means that, in a multithreaded application, the event handler will always be run on the user interface thread.

We will modify our control so that it generates a custom event, called `FocusableControlEvent`. This will allow us to send additional information with the event, such as whether the event was generated with the mouse, the spacebar, the Enter key, or the mnemonic key.

To start, re-create the `FocusableControl` project and name the class `FocusableControlWithEvents`. First, we need an `EventArgs`-derived class that can supply custom information to the event handlers for the event. We will call the class `FocusableControlEventsArgs`. Here is its definition:

```
public class FocusableControlEventsArgs : EventArgs
{
    public enum FocusableControlEventType
    {
        Mouse,
        SpaceBar,
        Enter,
        Mnemonic
    }

    private FocusableControlEventType eventType;

    public FocusableControlEventType EventType
    {
        get
        {

```

```
Pro NET 2.0 Graphics Programming
    return eventType;
}
set
{
    eventType = value;
}
}
```

```
public FocusableControlEvents(FocusableControlEventArgs et)
{
    eventType = et;
}
}
```

In this code, we define an enum that describes the possible reasons why the event might have been raised, and define a property in the FocusableControlEvents class that sets or retrieves an instance of this enum as the EventType.

Next, we define the delegate type that will represent the event. Add this delegate definition to the namespace:

```
public delegate void FocusableControlEventHandler(
    object sender, FocusableControlEvents e);
```

It is important that you add this class and delegate to the namespace after the FocusableControlWithEvents class. If you add the code in the namespace before the FocusableControlWithEvents class, Visual Studio .NET gets confused and will no longer be able to put up the Design window. We need the Design window in order to use the Properties window to create event handlers.

Next, modify the declaration of the FocusableControl class to include the event type FocusableControlEventHandler, a public member variable named FocusableControlClick:

```
public class FocusableControlWithEvents : System.Windows.Forms.Control
{
    private Color focusColor = Color.LightBlue;
    public event FocusableControlEventHandler FocusableControlClick;
```

You've just seen the entire infrastructure necessary to generate events. Now, let's modify the control to raise the events at the appropriate places, instead of putting up the dialog box.

Modify the ProcessMnemonic() method by adding the following:

```
protected override bool ProcessMnemonic(char charCode)
{
    if (Enabled && Visible && IsMnemonic(charCode, this.Text))
    {
        // Perform action associated with mnemonic.
        // Moves focus to our control.
        Focus();
        Graphics g = this.CreateGraphics();
        drawButton(g);
        g.Dispose();
        // Raise the event
        FocusableControlEvents args = new FocusableControlEvents(
            FocusableControlEvents.FocusableControlEventType.Mnemonic);
        if (FocusableControlClick != null)
            FocusableControlClick(this, args);
        return true;
    }
    return false;
}
```

Note that this code raises the event. Whether any code is run in response to that depends on whether any control opts to handle that event. We won't write an event handler in the FocusableControlWithEvents class itself, but when we place the control in a test form, we will have the test form implement a handler.

It is important that we test FocusableControlClick and make sure that it is not null. It will be null until an event handler is added to it. This will prevent our control from generating an exception in the case where the developer using the control puts the control on a form, doesn't write an event handler for the control, runs the test application, and clicks the control.

Note When implementing other events, verifying that the delegate is not null is even more important. If you implement an event that fires during the Paint event for the custom control, you cannot even bring up the Design window without an exception unless you verify that the delegate is not null before calling it.

Now modify the Click event as follows:

```
private void FocusableControl_Click(object sender, System.EventArgs e)
{
    Focus();
    // Generate the event
    FocusableEventArgs args = new FocusableEventArgs(
        FocusableEventArgs.FocusableControlEventType.Mouse);
    if (FocusableControlClick != null)
        FocusableControlClick(this, args);
}
```

The KeyDown event contains the code for generating the final two types of events, SpaceBar and Enter:

```
private void FocusableControl_KeyDown(
    object sender,
    System.Windows.Forms.KeyEventArgs e)
{
    if (e.Alt == false &&
        e.Control == false &&
        e.Shift == false &&
        (e.KeyCode == Keys.Space || e.KeyCode == Keys.Enter))
    {
        // Generate the event
        FocusableEventArgs args;
        if (e.KeyCode == Keys.Space)
            args = new FocusableEventArgs(
                FocusableEventArgs.FocusableControlEventType.SpaceBar);
        else
            args = new FocusableEventArgs(
                FocusableEventArgs.FocusableControlEventType.Enter);
        if (FocusableControlClick != null)
            FocusableControlClick(this, args);
        e.Handled = true;
        return;
    }
}
```

Next, compile the custom control. We need to modify our test application to handle our new event, but before the event will be available in the Properties window, you need to compile the custom control.

Testing the Control with a Custom Event

For the test application, create a Windows Forms application as before, and insert the same code into it as for the TestFocusableControl application you created for the previous example, with one exception; instead of adding a FocusableControl control to the form, add a FocusableControlWithEvents control.

Then add a handler for the FocusableControlClick event. As usual, select the control in the Design view and double-click the FocusableControlClick event in the Properties window. Visual Studio .NET will create an empty event handler and set up the wiring for the event handler in the InitializeComponent() method. Modify the event handler like so:

```
private void focusableControlWithEvents1_FocusableControlClick(
```

```

    object sender, FocusableControl.FocusableControlEvents e)
{
    MessageBox.Show("Source of event is: " + e.EventType);
}

```

Compile and run the application. You can now operate the control so that it generates its event in the four possible ways.

 Previous

Next 

 Previous

Next 

Deriving from an Existing Control

Another approach to building a custom control is to derive from an existing Windows Forms control. This avenue allows you to do the following:

- Make use of the existing functionality of a control.
- Extend the control by adding new properties.
- Change the functionality by handling some of the events of the control.
- Extend the control by generating new events.

There are several scenarios where this approach works well. For example, you can do domain analysis on validations required in your application. After you determine the set of validations that are possible, you can write a single parameterized custom control that is derived from the TextBox control. You can specify the data validations to be applied to each instance of your custom TextBox control via properties.

As another example, you could derive from the DataGrid control, and make an elaborate control similar to a spreadsheet. Perhaps the data would come from a DataSet, with an addition of some calculations to compute additional values. The calculations could be specified via properties.

A more domain-specific application might be a situation where you have a large number of list boxes or combination boxes, all of which display one or another subset of the same list of values. You could derive from the ListBox control, and add a property or two that would allow the user of the control to indicate which subset should populate the ListBox. In a similar vein, you might derive classes from TreeView and TreeNode controls to implement special tree views, such as a tree view used to navigate through the file system.

Creating a Custom Control Derived from a TextBox Control

As an example, we will derive from the TextBox control, and add a property that allows us to specify a simple validation. We will design a text box that is supposed to hold a numeric value, and performs data validation to make sure that its contents are in a given range. The upper and lower bounds of this range are defined by properties, and can be set by client code. If the client code sets the upper range to be less than the lower range, then the control will check that its contents lie *outside* the range. [Table 11-5](#) lists the actual limits we will use in our test application.

Table 11-5: NumberTextBox Properties

Lower Limit	Upper Limit	Meaning
0.0	100.0	Text in TextBox must be a number between 0.0 and 100.0
200.0	0.0	Text in TextBox must be a number that is either less than 0 or greater than 200.0

If, while validating, the control detects that its contents are out of range (or not a number at all), it will change its own background color to yellow. The control will also make available a Boolean property, `ContentsInRange`, which indicates whether it contains valid data, and which can be used by client code to determine whether to take other actions.

In the example, we will place a couple of these controls on a form and have the form display an error

symbol if the contents of the boxes are out of range. Hovering the mouse over this error symbol puts up a tooltip displaying an error message. The error symbol and associated tooltip are supplied not by our `NumberTextBox` class, but by the `System.Windows.Forms.ErrorProvider` class, provided as part of the .NET Framework class library.

The full running code will look like [Figure 11-11](#).



Figure 11-11: Using the `NumberTextBox` control

In [Figure 11-11](#), the first TextBox contains valid contents (as per [Table 11-5](#), it accepts a number between 0 and 100), while the second TextBox contains an invalid number. Validation is performed when the `Validating` event is raised for each `NumberTextBox` control. The `Validating` event is raised when the control loses the focus.

Note This is a very simple example designed to demonstrate the principles of deriving from an existing control. Obviously, in a commercial application, you probably wouldn't bother writing code to derive from an existing control unless you had some much more significant feature to add to the control. For minor features, it is usually easier to just use the existing control and manipulate it by methods in the containing form or control. Also, if you do want a control that allows the user to type in only numbers, you may find the existing `System.Windows.Forms.NumericUpDown` control fits your needs better.

To begin, create a new Windows Control Library in Visual Studio .NET and name it `NumberTextBox`. In the Solution Explorer, right-click `UserControl1.cs` and rename it to `NumberTextBox.cs`. Then, change the base class of the `NumberTextBox` class to the `System.Windows.Forms.TextBox` class.

We are going to add two string properties to the `TextBox` control: `RangeText` and `ValidationErrorMessage`. The `RangeText` property contains the low and high limits of the allowed number range separated by a comma. We are choosing to supply this property as a single string, rather than as two separate numeric properties, because we want the two limits of the range to be settable as one single property. This isn't particularly important at the moment, but will become important in the [next chapter](#) when we develop this example further to add more design-time support to it. The `ValidationErrorMessage` property will return a string describing the problem if the text in the control is out of range. This property is read-only.

However, before we start editing the code for the control, we will define a class that represents the exception that will be thrown if any client code attempts to set an invalid value for the `RangeText`, for example, a nonnumeric string. So, at the bottom of the namespace, add this class to the file:

```
public class InvalidRangeTextException : ApplicationException
{
    public InvalidRangeTextException(string s) : base(s)
    {
    }
}
```

It is important that this class be added at the *bottom* of the namespace. Visual Studio .NET requires that designers use the first class in the file. If you add this class to the beginning of the namespace, Visual Studio .NET will not be able to open the Design window for your custom control.

Now we will edit the control itself. First, add three private member variables to the `NumberTextBox` class. These define the range in both text and numeric form, and set default values:

```
private string rangeText = "0,100";
private double low = 0, high = 100;
```

Next, we will define the property that allows the range to be set or examined. Notice that we use the `DefaultValue` attribute to inform Visual Studio .NET of the default value:

```
[DefaultValue("0,100")]
public string RangeText
```

```

    {
        get
        {
            return rangeText;
        }
        set
        {
            if (value == null)
                throw new InvalidRangeTextException("Range text cannot be null");
            string[] v = ((string)value).Split(new char[] { ',' });
            if (v.Length != 2)
                throw new InvalidRangeTextException(
                    "Range text should have format <f>,<f> where
                     <f>=floating-point number");

            try
            {
                low = double.Parse(v[0]);
                high = double.Parse(v[1]);
            }
            catch (FormatException)
            {
                throw new InvalidRangeTextException(
                    "Validation parameters should have form <f>,<f> where
                     <f>=floating point number");
            }
            rangeText = value;
        }
    }
}

```

The `get` accessor for this property is fairly trivial. The `set` accessor is more complex, since we need to verify that the text to which the range is being set is of the form `<number>,<number>`, and throw an exception if it isn't. The code does this and also stores the limits in the fields `low` and `high`, so we can easily validate data later on.

Note You may wonder what happens if you try to set the string to an invalid value in the Properties window. In fact, when setting a property, Visual Studio .NET actually calls the `set` accessor against the control. As you will be able to verify when we have completed the example, Visual Studio .NET will automatically catch any exception and display a dialog box telling you the value is invalid.

We next need to add the property that tells us whether the text typed into the `TextBox` is in range:

```

[Browsable(false)]
public bool ContentsInRange
{
    get
    {
        double textBoxValue;

        // Check value is a number
        try
        {
            textBoxValue = Double.Parse(this.Text);
        }
        catch (FormatException)
        {
            return false;
        }

        // If high < low, then check number is OUT of range
        if (high < low)
        {
            if(textBoxValue < low && textBoxValue > high)
                return false;
            else

```

```
    }  
    // If high >= low, then check number is IN range  
    if (textBoxValue < low || textBoxValue > high)  
        return false;  
    return true;  
}  
}
```

The first item of interest in this code is that we apply a new attribute, `Browsable`, to it. I mentioned earlier in the chapter that applying this attribute and setting it to `false` will prevent Visual Studio .NET from displaying this property in the Properties window, but this is the first time you have seen this attribute in action. Clearly, testing whether the contents are in range is a purely runtime action, so it is pointless displaying this property in the Properties window at design-time. If we didn't apply this attribute, the property would be displayed, although it would be grayed out, since it is a read-only property.

As far as implementation of this property is concerned, we first try to convert the contents of the TextBox value to a double. If the TextBox contains a string like "hello," this will throw an exception, which we must then catch. Otherwise, we can check if the number is in the required range and return `true` if it is.

The final property we need to code is the error message to be returned if the contents are out of range:

```
[Browsable(false)]  
public string ValidationErrorMessage  
{  
    get  
    {  
        if (ContentsInRange)  
            return "";  
        if (high >= low)  
            return "Value must lie between " + low.ToString() +  
                   " and " + high.ToString();  
        else  
            return "Value must NOT lie between " + high.ToString() +  
                   " and " + low.ToString();  
    }  
}
```

Note that this property returns an empty string if the contents are valid, since clearly in this case there is no error message to be displayed. We also set the `Browsable` attribute to `false` to prevent display of this attribute in the Properties window.

The final bit of code we need to add is a `Validating` event handler that turns the background of the TextBox yellow if the contents are invalid (or returns it to white if the contents are correct). Click on the Design window, press F4 to open the Properties window, and click the Events button in the toolbar. Locate the `Validating` event and double-click it to create an empty event handler for it.

Modify the event handler to add code to perform the validation, like so:

```
private void NumberTextBox_Validating(  
    object sender, System.ComponentModel.CancelEventArgs e)  
{  
    if (ContentsInRange)  
    {  
        this.BackColor = Color.White;  
    }  
    else  
    {  
        this.BackColor = Color.Yellow;  
    }  
    e.Cancel = false;  
}
```

Notice that this event handler is passed a `CancelEventArgs` instance. `CancelEventArgs` is .NET's way of allowing a handler to cancel an event, by setting the `CancelEventArgs.Cancel` property to `true`. This mechanism is used, for example, to prevent the closing of an application if there is unsaved data. In the case of validating, you might opt to cancel the event if validation fails. However, we won't do

that here. The reason is that canceling a `Validating` event would prevent focus from leaving the control until the user corrects the contents of the text box. This may be appropriate for some applications, but here we want an application that displays an error symbol but still lets the user navigate around the form. Therefore, we always set `e.Cancel` to `false`, even if validation fails.

That's all there is to our custom control. Compile it and add it to the Toolbox window. All we have left to do is to build an application to test the control.

Testing the Derived Custom Control

Add a Windows Application to the project, call it `TestNumberTextBox`, and set it as the startup project.

In the Design window, place two `NumberTextBox` controls on the test application's form, along with two `Label` controls to label the text boxes. Use the Properties window to set the `RangeText` property for both `NumberTextBox` controls. For the first one, leave the `RangeText` property at its default value of `0,100`. For the second one, set it to `200,0`.

Next, use the Toolbox to add an `ErrorProvider` control to the form. Since the `ErrorProvider` control doesn't, by default, have an appearance, it will be represented by an icon below the form in the Design view. We will use this `ErrorProvider` control to display an error icon, if necessary, from the event handlers that we will write for the `Validation` events for each of the two `NumberTextBox` controls. For this case, we will use the same event handler for both controls, using the `sender` parameter passed to the handler to distinguish which control raised the event. So, in the Code view, type the following code in the `Form1` class:

```
private void validationTextBox_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    NumberTextBox.NumberTextBox tbSender = (NumberTextBox.NumberTextBox)sender;
    if (!tbSender.ContentsInRange)
    {
        this.errorProvider1.SetError(
            tbSender, tbSender.ValidationErrorMessage);
    }
    else
    {
        this.errorProvider1.SetError(tbSender, " ");
    }
}
```

This code uses the `ErrorProvider.SetError()` method, which takes a reference to a control and a string as a parameter. Provided the string is not empty, this method places that red error icon by the control and adds a tooltip containing the given error message. If the string is empty, `SetError()` removes any existing error icon from the control. Notice how we retrieve the error message from the `NumberTextBox` itself.

To attach this code to the `Validating` events, use the Properties window to locate the `Validating` event for each `NumberTextBox` in turn. Instead of double-clicking to add a new event handler method, click the arrow to open the drop-down list box, and select the name of the method that you just typed in, as shown in [Figure 11-12](#).

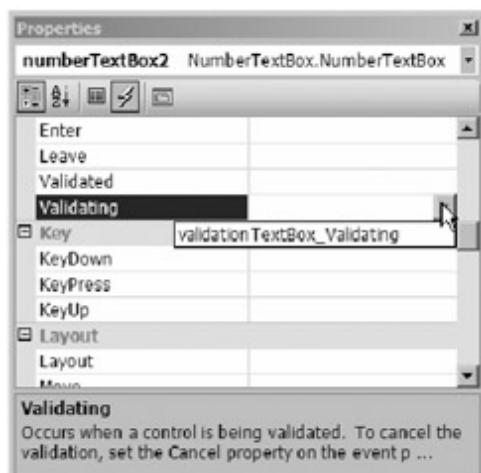


Figure 11-12: Setting the Validating event handler

Notice how the `Validating` event for each control now has two event handlers: one within the `NumberTextBox` class itself, which paints the control yellow if there is an error, and one within the containing form, which adds the error icon. This shows how event-based architecture allows different classes to add their own processing to any event.

The application is now ready to compile and run. Try out the `NumberTextBox` controls by entering a variety of values.

In the [next chapter](#), we will enhance this control in a couple of ways to make it more convenient to specify the validation parameters at design-time.

Previous

Next

Previous

Next

Building Composite Custom Controls

Another way to build a custom control is to compose it of any number of constituent controls. This approach is useful in several situations. For example, if you were building an accounting application, you would want to implement some standard behavior for entering transaction distributions. Or you might want to implement a filtering list box. You could implement a custom control that has the same user interface as a standard list box, with the addition of a text box that allows you to enter filtering information. As you enter filtering information into the text box, you update the contents of the list box to reflect the filter. If you were building a database application, you would want to have the navigation and database operation controls be identical for all windows. This can be a mechanism to enforce consistency across a development team or throughout an entire company.

You can declare properties for your composite control, which allows you to parameterize at a higher level of abstraction. As an example, suppose you had several buttons that navigate through and operate on a database. You could have a property that allows you to define one of the following options:

- This window is allowed to add, change, delete, and search for records.
- This window is allowed to only change and search for records.
- This window is allowed to only search for records.

Based on the value of this property, you could hide controls, move them around, change their text, and so on. A lot of functionality would be encapsulated in a single property.

You can add your own high-level events. You can handle events of the constituent controls, and if so desired, bubble them up to the user of the composite custom control.

Creating a Composite Control

To demonstrate this technique, we will build a custom control that contains four buttons. We will have a property that contains the database table name, and change the text of the buttons per this property. We will define an event that indicates whether the user asked to add, change, delete, or search.

Create a new Windows Control Library named `DataButtons` and rename `UserControl1.cs` to `DataButtons.cs`. Click Yes in the next dialog box to change the name of the class to `DataButtons`.

As always, when Visual Studio .NET builds a new project based on the Windows Control Library template, it defines the custom control with `UserControl` as the base class. In this case, that's what we want.

Switch to the Design view and resize the custom control, making it shorter and wider so that it can accommodate four Button controls placed horizontally across it. Drag four Button controls onto the custom control, and position them as shown in [Figure 11-13](#).



Figure 11-13: Layout of the DataButtons control

Set the properties for the Button controls as shown in [Table 11-6](#).

Table 11-6: Property Values for the DataButtons Control

Property	Button 1	Button 2	Button 3	Button 4
Text	Add	Change	Delete	Search
Name	btnAdd	btnChange	btnDelete	btnSearch

Within the `DataButtons` namespace, declare the following class and delegate for the event that our control will generate. As noted earlier, with Visual Studio .NET 2005, you must declare this class in the namespace *after* the declaration of the `DataButtons` class. If you declare it before the `DataButtons` class, Visual Studio .NET no longer understands that the control inherits from `UserControl`, and it will not allow you to manipulate the control in the Design window.

```
public class DataButtonsEventArgs : EventArgs
{
    public enum DataButtonsEventType
    {
        Add,
        Change,
        Delete,
        Search
    }

    private DataButtonsEventType eventType;

    public DataButtonsEventType EventType
    {
        get
        {
            return eventType;
        }
        set
        {
            eventType = value;
        }
    }

    public DataButtonsEventArgs(DataButtonsEventType et)
    {
        eventType = et;
    }
}
```

```
    }  
}  
  
public delegate void DataButtonsEventHandler(  
    object sender, DataButtonsEventArgs e);
```

Add the following member variables to the DataButtons class:

```
private string tableName;  
public event DataButtonsEventHandler DataButtonsEvent;
```

Add the following public property to the DataButtons class:

```
public string TableName  
{  
    get  
    {  
        return tableName;  
    }  
    set  
    {  
        tableName = value;  
        btnAdd.Text = "Add " + tableName;  
        btnChange.Text = "Change " + tableName;  
        btnDelete.Text = "Delete " + tableName;  
        btnSearch.Text = "Search " + tableName;  
    }  
}
```

In the Design window, double-click the Add button and add the following event handler:

```
private void btnAdd_Click(object sender, System.EventArgs e)  
{  
    // Generate the event  
    DataButtonsEventArgs args = new DataButtonsEventArgs(  
        DataButtonsEventArgs.DataButtonsEventType.Add);  
    if (DataButtonsEvent != null)  
        DataButtonsEvent(this, args);  
}
```

Similarly, double-click the Change button and modify the event handler:

```
private void btnChange_Click(object sender, System.EventArgs e)  
{  
    // Generate the event  
    DataButtonsEventArgs args = new DataButtonsEventArgs(  
        DataButtonsEventArgs.DataButtonsEventType.Change);  
    if (DataButtonsEvent != null)  
        DataButtonsEvent(this, args);  
}
```

Likewise, for the Delete button, add the following:

```
private void btnDelete_Click(object sender, System.EventArgs e)  
{  
    // Generate the event  
    DataButtonsEventArgs args = new DataButtonsEventArgs(  
        DataButtonsEventArgs.DataButtonsEventType.Delete);  
    if (DataButtonsEvent != null)  
        DataButtonsEvent(this, args);  
}
```

And finally, add the following for the Search button:

```
private void btnSearch_Click(object sender, System.EventArgs e)  
{  
    // Generate the event  
    DataButtonsEventArgs args = new DataButtonsEventArgs(  
        DataButtonsEventArgs.DataButtonsEventType.Search);  
    if (DataButtonsEvent != null)
```

```
    DataButtonsEvent(this, args);  
}
```

Compile the control.

Testing the Composite Control

Add a new Windows Application to the current solution and call it `TestDataButtons`. As per the usual procedure, set it as the startup project.

Drag a `DataButton` control from the Toolbox onto the form, resize it so you can see all of the buttons, and set the `TableName` property to `Customers`. Notice that the text of the buttons changes.

Add a couple of `TextBox` controls to the form, so that there are controls to which you can tab from our buttons. Select the `DataButtons` control in the form and open the Properties window. Click the Events button in the toolbar. Scroll to find the `DataButtonsEvent` and double-click it to create an empty event handler. Modify the event handler as follows:

```
private void dataButtons1_DataButtonsEvent(  
    object sender, DataButtons.DataButtonsEventArgs e)  
{  
    MessageBox.Show(e.EventType.ToString());  
}
```

Compile and run the application. You'll see that you can tab between the controls, click the buttons, and so on.

We have now completed working through examples of different custom controls. In the last section of this chapter, we will examine some of the principles of designing good controls.

 Previous

Next 

 Previous

Next 

Designing Components and Custom Controls

When it comes time to do the architectural design of your planned components and controls, there are two questions to ask:

- What components and controls should you build?
- What should the programming interface be with your components and controls?

The question about your programming interface can be further divided into three questions:

- What properties do you need?
- What methods do you need?
- What events should you raise?

In this section, I am going to present a methodology that can help answer these questions.

Using Nonprocedural Constructs

All programmers are familiar with using both procedural code and nonprocedural resources when building their applications. Windows resources that define dialog boxes, menus, and strings are good examples of nonprocedural constructs.

Have you ever built an application where you initialized an array of structures, then used the initialized array to drive your program? When designed properly, this technique allows you to change the behavior of your program by changing the data in the array. You don't need to touch the procedural code. This is another example of using a nonprocedural construct. Many programmers have used this nonprocedural aspect of programming with Microsoft's tools. Visual Basic and Visual C++ have offered this feature for years.

When performing systems analysis of software systems, there are many ways to slice and dice the application. You can create UML class diagrams. You can construct UML sequence diagrams. You can create data flow diagrams. You can document the entire system using use cases, either in text or in diagrams. This division between procedural and nonprocedural constructs is just one more way to view the design of the system that can yield insights. It can lead to applications that are easier to build and are more maintainable.

This technique doesn't take the place of classic object-oriented design. There are certain aspects of software systems that must be analyzed classically. It is an analysis technique to be used in conjunction with other design methodologies. This design methodology is, however, particularly germane to designing components and custom controls.

When you examine a problem with an eye toward analyzing what should be implemented nonprocedurally and what should be implemented procedurally, you need to pay attention to a particular idea. Which parts of the application have a nonprocedural nature, and which parts have a procedural nature? If you implement something that is nonprocedural in nature with procedural code, the procedural code tends to obfuscate the description of the nonprocedural constructs. We have all seen a long section of code that creates a user interface. Although we can read the code and discern what the user interface is, it is hardly obvious. Yet, when we look at a user interface in a design window, and see the properties of each of the controls in a property window, the user interface is quite easy to see.

What is procedural in nature should be described procedurally, and what is nonprocedural in nature should be described nonprocedurally. When your tool (Visual Studio .NET and the .NET Framework in this case) doesn't provide nonprocedural constructs that match the abstractions in your application, it is far better to first build components that implement the nonprocedural constructs.

As a broad estimate, the typical programmer uses approximately 25% nonprocedural constructs and 75% procedural constructs. (Of course, you might have a different profile.) I have developed applications that are more than 80% nonprocedural constructs, and the results were great. Of course, before I developed the application, I needed to build the custom controls that implemented the nonprocedural constructs, and there was plenty of procedural code within those custom controls.

Note It is interesting that Visual Studio .NET takes your nonprocedural constructs (the properties of your components) and stores them in procedural code that is, by default, hidden from view. However, you don't need to concern yourself too much with the method that the IDE uses to store the properties of components. It is more important that the way you view and modify the properties is in an appropriate form, which is the tabular view in the Properties window.

The further you push the line toward nonprocedural code, the easier it is to build and maintain your application. The following are the main benefits of using nonprocedural constructs:

- **Better productivity:** When developing a system, if you first build better abstractions using nonprocedural constructs, you can then "assemble" the system. Changes in the specifications often don't require changes in procedural code.
- **Greater reliability:** Once you debug a particular nonprocedural construct, you can use that construct within the system with a minimum of debugging. Coverage testing is the process by which you verify that every procedural line of code has been executed, or if it hasn't been executed, you document why. You don't need to coverage-test the usage of nonprocedural constructs.
- **Improved maintainability:** It is far easier to document properties and the use of the properties than it is to document methods and their use. This makes it easier to transfer knowledge to maintenance programmers.

It is rare that an application designer can write the specifications for an application without making any mistakes. However, with properly constructed components and custom controls, in many cases, changes in the specification that happen later in the development process can be easily accommodated, in many cases, without changing any procedural code.

Often when building an application, it can be useful to build a component or custom control even when you are planning on using only a single instance of it. The reason is that this allows you to design a portion of your program that is data-driven. With a properly designed component, you can then change the behavior of your application by simply changing properties of the component. You don't need to touch the code.

Implementing Nonprocedural Constructs in Component Design

So, how does this nonprocedural philosophy apply to the process of making components and controls? There is a five-step process:

1. Look at the application and determine the abstractions that are candidates for implementation using nonprocedural constructs. In the case of user interfaces, do you have specialized validations that repeat throughout the application? Do you have repeating patterns of controls that occur in more than one window? You then document each of the abstractions and what the abstractions do.
2. Now that you know what the abstractions do, you need to figure out what you need to know in order to do what you need to do.
3. Design properties that contain what you need to know.
4. Implement the controls with appropriate properties.
5. Use those components to assemble your application.

I tend to err on the side of making more properties than I may actually need. I have never been sorry that I defined more properties, even if I didn't ever use a particular property to change the behavior of an application. However, I have been sorry that I didn't abstract a particular characteristic of a component and make it into a property.

Tip If you create a property that you suspect may not be used, you can always configure the property so that it will not show up in the Properties window in Visual Studio .NET. You will see how to do this when we discuss design-time configuration of components and custom controls in the [next chapter](#).

A somewhat simplistic way to describe this philosophy is:

- Make more custom controls and give them more properties.

If, to implement a particular piece of functionality, you have the choice of writing a method or creating some properties, the preference should be given to creating properties.

Don't write code to validate text boxes. Rather, design a custom control that has special properties (nonprocedural constructs) that allow you to specify the validation in a nonprocedural way. Don't write code to import data from or export data to a remote source. Rather, build a component that allows you to specify the import/export in a nonprocedural way. After your components are mature, you will find that you will use these components over and over again without modification in project after project.

Any time that you find yourself writing the same code over and over, this is a clue that there is an abstraction that you can generalize. More than this though, you should analyze your application and determine your abstractions ahead of time.

Of course, you cannot specify everything in a nonprocedural fashion. The very nature of what happens when the user clicks a button is procedural. The procedures should be short and sweet though. Short event handlers are easier to debug than long event handlers.

Designing Events

When designing events, the events that you need to implement are normally fairly obvious. A Button control needs a Click event. A TextBox control needs an event that tells you when the text changes.

There is one area that developers should watch out for when designing events, and there are some guiding principles that should come into play when designing these types of events:

- Tell the module using the control what is about to happen, before it happens. In other words, have an event that indicates that the focus is about to move.
- Give the module using the control some choice about what is about to happen. This is the concept of the refusible event. In the case of the .NET Framework, this is implemented by the CancelEventHandler delegate.
- Then you can take the action after the module using the control approves.

Not every event fits this model, and you might have a pair of events where the first event tells that something is about to happen and is refusible. The second event tells that something already happened

and is not refusible. This is the case of the `Validating` and `Validated` events of the `Control` class.

 PreviousNext  PreviousNext 

Summary

In this chapter, we progressively built more and more elaborate custom controls. From this foundation, you now have the skills necessary to build a wide variety of custom controls.

You saw that custom controls are components that have a visual aspect. They are components that have additional responsibilities, such as drawing themselves, handling events, and generating events. You also saw that there are three main types of custom controls that you can build:

- Built from scratch
- Derived from an existing control
- Assembled from multiple existing controls

You learned the mechanics of adding controls to the Toolbox, as well as how to add a reference to your custom control when building an application that uses the custom control.

Next, you learned about adding properties to controls and also saw that there are two methods of informing Visual Studio .NET of the default values: through an attribute that precedes the property or by the addition of two methods that Visual Studio .NET will call to get the default value and to reset the property to the default value.

Navigation is an important topic. Your custom controls should behave properly and have the same user interface cues as native controls. We covered the mechanics of allowing a control to gain focus and to have the correct focus cues.

Then we quickly reviewed delegates and events, and wrote a couple of controls that generate custom events.

Deriving custom controls from existing Windows Forms controls is a powerful technique to develop higher-level abstractions. As you learned in this chapter, this technique, used in combination with the technique of building composite custom controls, gives you a great deal of leverage.

Finally, I explained a particular philosophy of component design: that of declarative programming. You saw some of the benefits of making components and controls that are largely data-driven. Using this philosophy can sometimes help you to decide which components and controls to create, and to design the properties and methods for them.

There is more to the topic of developing custom controls. Specifically, you can enhance the design-time experience of the developer using your custom controls. In the [next chapter](#), we will explore the design-time capabilities in depth.

 PreviousNext 

 PreviousNext 

Chapter 12: Design-Time Support

Overview

In this chapter, we will examine some of the ways to make your controls integrate better with Visual Studio .NET, making it easier for other developers to use your controls in their code. You will learn how to provide design-time support in several ways:

- Enhance the way the properties of your control are edited in the Properties window.
- Allow for Visual Studio .NET to bring up a modal dialog box or drop-down control to edit values of properties in your control.
- Enhance the way that your control is displayed in the Design view.

This means, incidentally, that none of the code we develop in this chapter will have any effect on how a custom control behaves at runtime. Rather, it will improve the way that others are able to use your control at design-time, when they are building it into some other application. The concepts that we will develop are nevertheless extremely important. As a developer who uses Visual Studio .NET, you expect to be able to easily manipulate any controls that you place in your forms. Well, other developers will have similar expectations when it comes to manipulating the controls that you write. In this chapter, you will learn how you can satisfy those expectations.

 PreviousNext  PreviousNext 

Enhancing Design-Time Support

With most previous component technologies, some of the enhanced design-time support was embedded in the design tool, rather than the component. This meant that while you might have had a very complex custom control, you had a limited ability to affect how the design tool visually represented the control. Only certain controls—such as edit controls, buttons, and the like—for which the enhanced design-time support was built into the IDE, had a graphical representation that was representative of the actual control or allowed direct manipulation of its characteristics. In contrast, an elaborate custom control from a third party would be represented as a box in the IDE.

All of this has changed with the .NET Framework and Visual Studio .NET. Now, all aspects of the design-time representation of the control are embedded in the control, not the development environment. This is true for all controls, including the controls in the Windows Forms namespace. You can make the design-time representation as elaborate as you want it to be. Basically, you can improve the way properties are manipulated in the Properties window and improve the way a control as a whole is displayed in the Design view.

Editing Properties in the Properties Window

In most cases, when you build custom components and controls, you can use them in an IDE such as Visual Studio .NET. You can add the controls to the Toolbox window and drag them from the Toolbox to the Design window for a form. You can select the control on the form and edit its properties.

However, if one of your properties is a class or structure, unless certain provisions are made, you cannot edit that property. For this type of property, at the very least, you'll want to allow for creating a string from the structure, and creating the structure from a string. For example, if you declare a property of type `Point`, you can edit the property as a string by entering the two components of the `Point` (width and height), separated by a comma. In addition, with the `Point` structure, you can treat it as a *composite property*, and expand the property and edit the width and height separately.

In addition, you can make a number of enhancements so that editing complex properties is much more convenient. You may design a custom GUI for your property, so that the developer using the property

can use some form of direct manipulation to edit the property. You may put up a modal dialog box with a number of fields, check boxes, radio buttons, and so on to allow editing of a complex property in a more intuitive form.

The following is a list of the ways in which you can enhance design-time support for properties, from the simplest to the most elaborate:

- Assign default values to properties.
- Hide properties that aren't relevant at design-time so they don't appear in the Properties window.
- Assign properties to categories.
- Convert properties to and from strings using a `TypeConverter`.
- Create composite properties that you can expand and collapse using a `TypeConverter`.
- Create a modal dialog box in which you can edit a property.
- Create a drop-down window in which you can edit a property.

In the [previous chapter](#), you learned how to assign default values to properties and hide properties. By using the `Browsable` attribute, you can prevent properties from appearing at all. The `DefaultValue` attribute causes values to display in bold if they are not the default value. Let's quickly go over the rest of the design-time support list here.

Categorizing Properties

When you define a property, you can also define a category for it. When you open the Properties window in Visual Studio .NET, you have two viewing options: seeing the properties organized by category or seeing them sorted in alphabetical order. You switch between these two options via the toolbar at the top of the Properties window.

When the properties are categorized, you can collapse and expand categories in the Properties window. [Figure 12-1](#) shows properties sorted by category, with the Layout category collapsed.

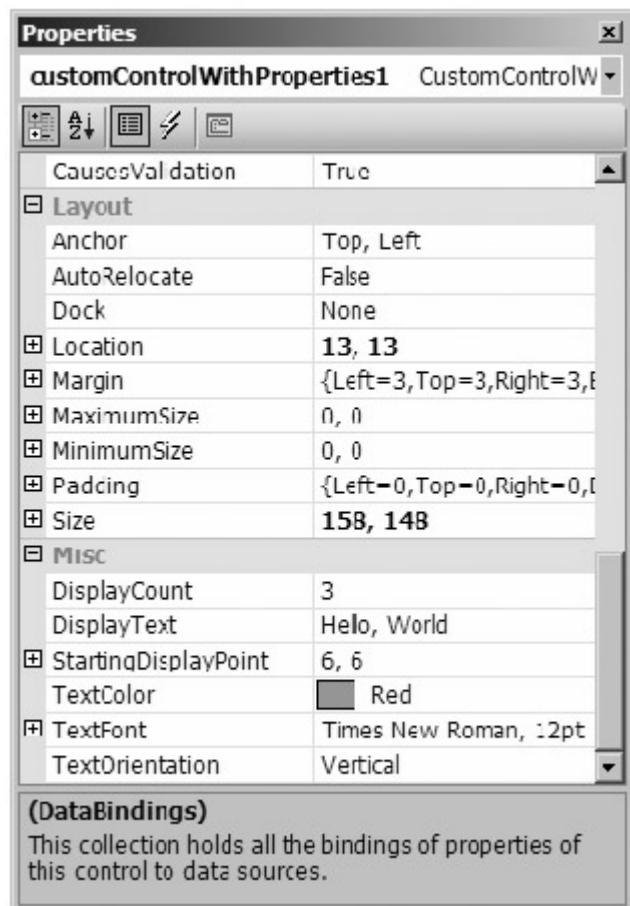


Figure 12-1: Properties sorted by categories

Expanding Composite Properties and Converting to Strings

Adding one more level to a hierarchical view in the Properties window, you can design *composite properties* that expand and collapse, but which can also be edited directly via *string representations*. This allows you to further organize properties for ease of editing. In [Figure 12-2](#), the StartingDisplayPoint property is a composite property that expands. This way, the developer can either edit the X and Y components of it independently or edit the string representation (6,6 in the example).

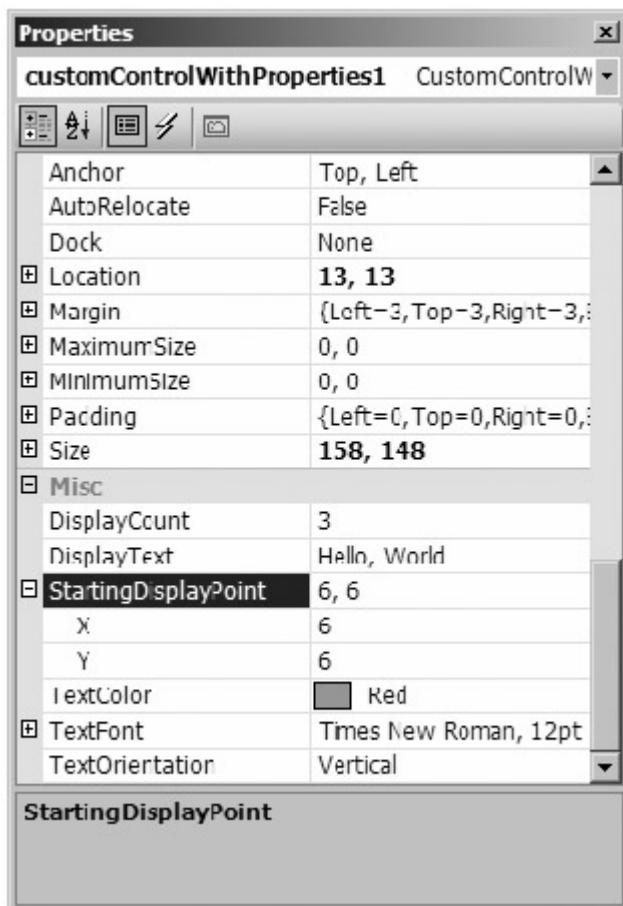


Figure 12-2: Expanded composite property

You will learn how to create properties such as `StartingDisplayPoint` in the "[Improving Editing in the Properties Window](#)" section later in this chapter.

Using Custom Property Editors

A wide variety of custom property editors are already defined for certain types within the .NET Framework:

- Font types are edited with a font selection dialog box.
- Colors are edited with a color picker.
- The `Anchor` property allows you to interactively select how a control is anchored to the sides of its containing control.
- The `bool` and `enum` properties are edited with a drop-down list, which is similar to a custom property editor.

These custom editors can either be put in the drop-down position or opened as modal dialog boxes.

Editing the Control in the Design View

For complex custom controls, you can enhance the way that the control is displayed on the design surface. For example, if you built a complex spreadsheet control, you might change the representation so that the developer using the control could change widths of columns through direct manipulation with the mouse.

As another example, some controls, such as those that group other controls together, may take up screen real estate but not have any appearance by default. In this case, you will want to at least draw a dotted line that represents the control, so that the developer using the control can manipulate it.

You will learn more about this aspect of improving controls toward the end of the chapter, in the "[Implementing a Custom Designer](#)" section.

 Previous

Next 

 Previous

Next 

Categorizing Properties and Events

In the examples in the [previous chapter](#), when we added properties and events to our custom controls, they were added to the Misc category of the Property window by default. The `Category` attribute will cause the property to be placed under the given named category.

As an example, consider the `CustomControlDefaultPropValues` example from [Chapter 11](#). That control has several properties related to the control's appearance, such as `DisplayText` and `DisplayCount`, so they should be in the Appearance category of the Properties window. You can have them appear there by adding the `Category` attribute. The code shown here is for the `DisplayText` attribute. The code is identical for the other attributes.

```
[DefaultValue("Hello, World"), Category("Appearance")]
public string DisplayText
{
    get
    {
        return displayText;
    }
    set
    {
        displayText = value;
        Invalidate();
    }
}
```

The code can alternatively be written as follows:

```
[DefaultValue("Hello, World")]
[Category("Appearance")]
public string DisplayText
{
    .
    .
    .
}
```

Now, when you open the Properties window, you see the attributes in the Appearance category, as shown in [Figure 12-3](#).

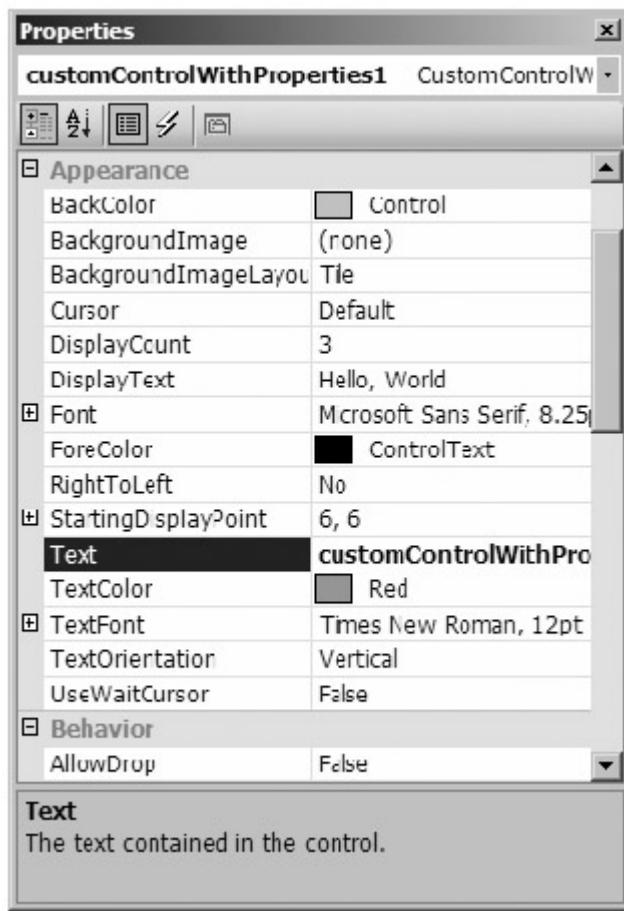


Figure 12-3: Properties assigned to a category

Within each category, the properties are sorted by property name. If desired, you can change this by adding the `ParenthesizePropertyName` attribute, which forces the property to go to the top of the category, besides putting its name in parentheses.

Note that in this example, the properties all had another category, `DefaultValue`, already applied. This is not necessary; the two attributes are independent, so you can use the `Category` attribute by itself. However, it's good programming practice to supply both a `DefaultValue` and a `Category` to all browsable controls, to make it easier for developers to use them.

Using the `Category` attribute, you can place each property in whatever category you wish. You can either use one of the existing categories (such as `Appearance`) or make up your own simply through your choice of the string you supply to the `Category` attribute.

Previous

Next

Previous

Next

Improving Editing in the Properties Window

We are now going to return to the `NumberTextBox` custom control introduced in the [previous chapter](#) and add improved design-time support to it. In this section, we will focus on how the properties of the control are edited in the Properties window.

In the control's current form, the possible range of numbers are entered as a string, as in `0,100`, so the developer (or any client code) can modify both values at the same time. An improvement would be to allow the values to be edited together or individually. That's what we are going to achieve in this section. The custom control will look and behave exactly the same as the `NumberTextBox` example, but it will have enhanced editing facilities in the Properties window, as shown in [Figure 12-4](#).

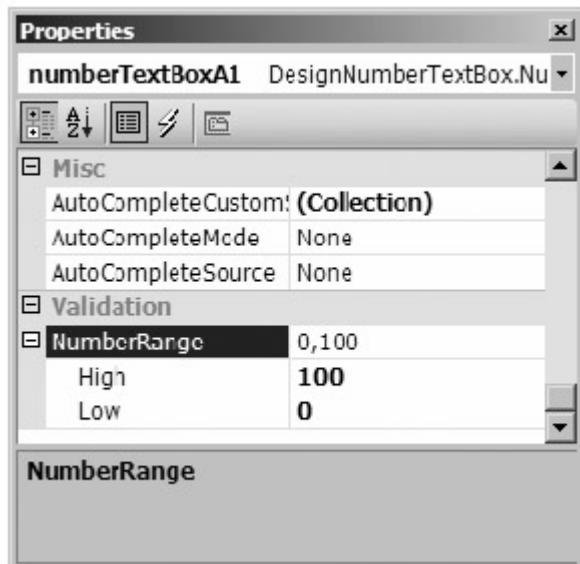


Figure 12-4: The NumberRange composite property

As you can see, the `NumberRange` property is expandable so that you can edit it, either by typing in the comma-separated values as a string or by typing in the individual Low and High components.

Additionally, this example will demonstrate how to allow IDE editing of properties not available by default. Up until now, you have been able to edit only properties that are of certain data types in the Properties window, such as a string or point. If one of your custom controls implemented a property that was of a type you had defined, by default, Visual Studio .NET will not allow you to edit that property in the Properties window. This is because the IDE has no way of knowing what your intentions might be regarding how a developer should be able to edit that type.

Using the techniques you'll learn about in this section, you can make your properties editable in the Properties window. In fact, in this example, `NumberRange` is an instance of a struct that we will define, also called `NumberRange`. As [Figure 12-4](#) shows, Visual Studio .NET allows full editing functionality for the possible values of this struct. As you will see when we write the code for the example, adding this functionality requires two players:

- The struct (or class) that represents the property
- A type-converter class that is derived from `System.ComponentModel.TypeConverter` and controls conversion of the property type to and from a known type, such as `string`

This custom control will not have much code in common with the previous version of `NumberTextBox`, so let's start a new project from scratch. Create a new Windows Control Library project, and name it `DesignNumberTextBox`.

Note For this example, we will use the name `DesignNumberTextBox` for the solution and project, but the control itself will be named `NumberTextBoxA`, to distinguish it from the `NumberTextBox` class of the control we created in [Chapter 11](#). So, if you are creating this project as you read, you should make the appropriate changes to your C# files. Or, as with all the examples in this book, you can simply download the complete sample from the Downloads section of the Apress web site at www.apress.com.

Defining the NumberRange Struct

Our new structure, `NumberRange`, will perform data validation. This is the struct that holds the low and high limits and performs the data validation. This structure replaces the string that we used in the previous version of the `NumberTextBox` control. In a real-world application, you would make this structure quite elaborate, with many types of data validations in it. You would have an enumeration for all the types of validations, and a number of parameters to be used for each type of validation. For our example though, the validation is a simple test of whether a given number lies within the specified range.

The definition of the `NumberRange` struct starts off as follows. We define a couple of member fields that

specify the low and high limits of the range, along with corresponding properties. Add this structure to the DesignNumberTextBox namespace, but add it *after* the NumberTextBoxA class:

```
public struct NumberRange
{
    private double low;
    private double high;

    public NumberRange(double low, double high)
    {
        this.low = low;
        this.high = high;
    }

    public double Low
    {
        get
        {
            return low;
        }
        set
        {
            low = value;
        }
    }

    public double High
    {
        get
        {
            return high;
        }
        set
        {
            high = value;
        }
    }
}
```

Since this struct will be responsible for validating data, we also need to add this method:

```
public bool Validate(string s)
{
    double val;
    // Check value is a number
    try
    {
        val = Double.Parse(s);
    }
    catch (FormatException)
    {
        return false;
    }

    // If high < low then check number is OUT of range
    if (high < low)
    {
        if(val < low && val > high)
            return false;
        else
            return true;
    }

    // If high >= low then check number is IN range
    if (val < low || val > high)
        return false;
}
```

```

    return true;
}
}
```

This method does exactly the same thing as the `ContentsInRange` property of the `NumberTextBox` class in the example from the [previous chapter](#), except that due to the need to pass in the string as a parameter, it is now a method rather than a property. Recall also from the [previous chapter](#) that if the high limit is lower than the low limit, we interpret this as a requirement that the number should lie outside, rather than inside, the specified range.

At this point, we have a complete struct to perform data validation. What we don't have yet is any ability for Visual Studio .NET to allow editing of the values of this type, which is the whole point of this example. This aspect comes by adding the following attribute to the struct:

```
[TypeConverter(typeof(NumberRangeConverter))]
public struct NumberRange
{
    .
    .
    .
    .
}
```

Up until now, we haven't covered much new territory. Now comes the fun part.

Creating the TypeConverter-Derived Class

To allow developers using our control to edit this new property, we need to create a `TypeConverter` for the class. The `TypeConverter` class does two things for us. First, it allows the developers to edit the property as a string. The way that the developer edits the string will be the same as in the previous versions of this custom control. In other words, the low and high parameters are viewed and entered as a comma-separated string. However, our parameters will be stored in our own class. The `TypeConverter` also enables our property to be a composite property, which we can expand and collapse in the Properties window.

Both functions that the `TypeConverter` serves are important. Some developers will prefer to edit the property as a string, rather than expanding the property, and even if a developer prefers to expand the property to edit it, the conversion to a string data type is important because the conversion to string is used to provide the value of the property in the right column of the Properties window when the property is not expanded.

The `NumberRangeConverter` class derives from the `TypeConverter` class, which implements four main areas of functionality:

- Type conversion between your property data type and the string data type
- Type conversion from your property data type to an `InstanceDescriptor` object
- Subproperties, so that the developer using the control can expand/contract the property
- Re-creation of the data type when the data type is immutable

The first area of functionality is type conversion between the `NumberRange` structure and a string. We convert to and from the string data type so that the Properties window can display a string in the right column for the composite property itself. Conversion to a string allows display of the property, while conversion from a string allows developers to edit the composite property, perhaps as a comma-delimited string, as implemented in this example.

The second area of functionality of the `NumberRangeConverter` class is to convert from our property data type to an instance of the `InstanceDescriptor` class. Visual Studio .NET uses an `InstanceDescriptor` object for generation of code for modifying the `NumberRange` property. The `InstanceDescriptor` object describes the constructor signature of the `NumberRange` class, and from the constructor signature, the code generator can write instantiation code in C#. Note that you need to do a type conversion in only one direction: from `NumberRange` to `InstanceDescriptor`.

Although these areas of functionality have different purposes, they are implemented in the same methods, so we will consider them together as far as the code is concerned.

Using Type Conversion Methods

You must implement four methods to effect type conversion:

- `CanConvertFrom()`: Tests against the `Type` argument, and returns `true` or `false` based on the type.
- `CanConvertTo()`: Tests against the `Type` argument, and returns `true` or `false` based on the type.
- `ConvertFrom()`: Converts the `value` argument (of type `object`), and returns an object of your parameter type (`NumberRange` in this case).
- `ConvertTo()`: Converts the `value` argument, which is declared as type `object`, but is actually of type `NumberRange`, and returns an object of the destination type, which will be either of type `string` or type `InstanceDescriptor`. It will be only one of these two types because these are the two types we specified in the `CanConvertTo()` method.

These methods are all defined in the `System.ComponentModel.TypeConverter` class, and you need to override these methods to provide your own type conversion ability.

All four of these methods take an `ITypeDescriptorContext` as an argument. From this argument, you can determine the actual control that instigated the type-conversion request, and you can determine the container of that control, just in case you need that information in order to perform the conversion. In our example, we don't care what the context is; our conversion is not affected by it.

In addition, the `CanConvertTo()` and `ConvertTo()` methods take an argument of type `CultureInfo`. You could use this information to influence the conversion. For example, you could use this argument to modify the conversion of a decimal to a string, so that the conversion happened correctly based on the `CultureInfo` value. In our example, we don't care about this argument.

Let's start with the class definition and the `CanConvertFrom()` method. All this method needs to do is return a value to indicate that our class will allow conversion from a string, as well as any conversions supported by the base class. Again, add this class to the `DesignNumberTextBox` namespace *after* the `NumberTextBoxA` class:

```
public class NumberRangeConverter : System.ComponentModel.TypeConverter
{
    public override bool CanConvertFrom(
        ITypeDescriptorContext context, Type sourceType)
    {
        if (sourceType == typeof(string))
            return true;
        return base.CanConvertFrom(context, sourceType);
    }
}
```

Next is the method that actually does the conversion from a string. The code here is very similar to that for the previous version of the `NumberTextBox` control in [Chapter 11](#). We simply identify where the comma is in the string, and assume that the substrings separated by the comma are the low and high values:

```
public override object ConvertFrom(
    ITypeDescriptorContext context, CultureInfo culture, object value)
{
    if (value is string)
    {
        string[] v = ((string)value).Split(new char[] { ',', ',' });
        if (v.Length != 2)
            throw new ArgumentException(
                "number range string must be of form <low limit>,<high limit>");
        NumberRange nr =
            new NumberRange(double.Parse(v[0]), double.Parse(v[1]));
        return nr;
    }
    return base.ConvertFrom(context, culture, value);
}
```

Now we need to do the same thing, but converting to a value. The `CanConvertTo()` method needs to return a value to indicate we can convert to either a string or an instance descriptor, or to any type supported by the base class:

```
public override bool CanConvertTo(
    ITypeDescriptorContext context, Type destinationType)
{
    if (destinationType == typeof(string))
        return true;
    if (destinationType == typeof(InstanceDescriptor))
        return true;
    return base.CanConvertTo(context, destinationType);
}
```

And similarly, having informed Visual Studio .NET that we can do these conversions, we need to implement the method to actually perform the conversions:

```
public override object ConvertTo(
    ITypeDescriptorContext context, CultureInfo culture, object value,
    Type destinationType)
{
    if (destinationType == typeof(string))
    {
        NumberRange nr = (NumberRange)value;
        return nr.Low + "," + nr.High;
    }
    if (destinationType == typeof(InstanceDescriptor))
    {
        NumberRange nr = (NumberRange)value;

        // Specify that we should use the two-parameter constructor
        return new InstanceDescriptor(
            (typeof(NumberRange)).GetConstructor(
                new Type[2] {
                    typeof(double),
                    typeof(double)
                },
                new double[2] {
                    nr.Low,
                    nr.High
                }));
    }
    return base.ConvertTo(context, culture, value, destinationType);
}
```

The code for conversion to a string simply sticks the low and high values together in comma-delimited format. The conversion to an instance descriptor supplies information about the constructor of the type used for the property. Recall that `NumberRange` has a constructor that takes two `double` parameters, used to populate the `Low` and `High` properties:

```
public NumberRange(double low, double high)
{
    .
    .
    .
}
```

This code returns an `InstanceDescriptor` that contains that information in a format that Visual Studio .NET can understand. (For more information about the `InstanceDescriptor` class, see the MSDN documentation.)

Implementing Subproperties

The third area of functionality for a type converter is implementation of subproperties. To implement this,

we override two methods, `GetPropertiesSupported()` and `GetProperties()`. Our override of the `GetPropertiesSupported()` method simply returns true, to indicate that this custom class has properties. Our override of `GetProperties()` needs to return an object of type `PropertyDescriptorCollection`. There is a convenient method of the `TypeDescriptor` class that returns a `PropertyDescriptorCollection` if you pass in a type. This method uses reflection to get a list of all of the properties in our `NumberRange` data type. If we want to change the list of subproperties, we can modify the `PropertyDescriptorCollection` before we return it.

Here are the two overridden methods:

```
public override bool GetPropertiesSupported(
    ITypeDescriptorContext context)
{
    return true;
}

public override PropertyDescriptorCollection GetProperties(
    ITypeDescriptorContext context, object value, Attribute[] attributes)
{
    PropertyDescriptorCollection properties =
        TypeDescriptor.GetProperties(typeof(NumberRange), attributes);
    return properties;
}
```

The `GetProperties()` method takes an `ITypeDescriptorContext` as an argument. We could use this to determine the actual object that instigated the request.

In addition, `GetProperties()` takes an array of `Attribute` objects, to be used for filtering. The main purpose for this argument is to filter out all properties for which the `Browsable` attribute is set to `false`. As it turns out, there is a `GetProperties()` method in the `TypeDescriptor` class that also takes an array of attributes, for exactly the same purpose. We can pass the `Attribute` array to the method, and it takes care of the issue for us.

DEALING WITH IMMUTABLE CLASSES

An object of an immutable class (such as `System.String`) can never be modified once it has been created. This means that if you are editing the property values of an immutable class in the Properties window, Visual Studio .NET won't be able to modify the object. Instead, it will need to delete that object and create a new one with the new values.

To enable Visual Studio .NET to destroy and rebuild the object that contains the value of an immutable composite property, you need to implement two methods in the type converter class: `GetCreateInstanceSupported()` and `CreateInstance()`.

`GetCreateInstanceSupported()` should return a `bool` that indicates whether the type converter class can supply information about how to create a new instance of the property. You override this method to return `true`.

`CreateInstance()` actually returns the information needed to construct a property instance. You override this method and implement it as follows. This sample code describes the situation for the `NumberRange` class, whose constructor takes two parameters that are used to set the `Low` and `High` properties:

```
public override object CreateInstance(ITypeDescriptorContext
    context, IDictionary propertyValues)
{
    return new NumberRange((double)propertyValues["Low"],
        (double)propertyValues["High"]);
}
```

You can use the indexer of the `IDictionary` interface, passing in the names of your properties, then cast the properties to the correct type, then call the constructor of your `NumberRange` structure, and then return the newly created object.

You shouldn't implement this functionality unless your data type is immutable. The `NumberRange` class in the example presented in this chapter is not immutable, so these considerations don't apply

Creating the NumberTextBoxA Control

The next class to examine for our DesignNumberTextBox control is `NumberTextBoxA`, which derives from `System.Windows.Forms.TextBox` and represents the actual control we are developing. This class is similar to the `NumberTextBox` class from the [previous chapter](#), except that the validation is now controlled by the new `NumberRange` class.

We start off by declaring a member field to hold the `NumberRange` validator. We also define a static, read-only field that holds the default value for `NumberRange`. Having this static field means that if we decide to change the default value, we can do so easily.

```
public class NumberTextBoxA : System.Windows.Forms.TextBox
{
    private static readonly NumberRange defaultNumberRange =
        new NumberRange(0,100);
    private NumberRange numberRange = defaultNumberRange;
```

Following the private member variables are the properties themselves. These properties are similar to the properties that we have used in the previous custom control examples. One minor difference is that the `NumberRange` property returns a structure, as opposed to a variable such as a string or integer. We have also defined a suitable category for this property (using the `Category` attribute, as described earlier in this chapter):

```
[Category("Validation")]
public NumberRange NumberRange
{
    get
    {
        return numberRange;
    }
    set
    {
        numberRange = value;
    }
}
```

Next, we need to deal with informing Visual Studio .NET of the default value of the `NumberRange`. Since `NumberRange` is now a structure, rather than a string, we can no longer use the `DefaultValue` attribute to do this, so we will need to use the `Reset/ShouldSerialize` technique (explained in [Chapter 11](#)):

```
public void ResetNumberRange()
{
    NumberRange = defaultNumberRange;
}

public bool ShouldSerializeNumberRange()
{
    return ! NumberRange.Equals(defaultNumberRange);
}
```

Although validation is performed by the `NumberRange` class, we still want our control to expose a property indicating if the contents are valid. This is it:

```
[Browsable(false)]
public bool ContentsInRange
{
    get
    {
        return this.NumberRange.Validate(this.Text);
    }
}
```

Just as in the earlier example, we also have a property that returns a suitable validation error message:

```
[Browsable(false)]
public string ValidationErrorMessage
{
    get
    {
        if (ContentsInRange)
            return "";
        if (numberRange.High >= NumberRange.Low)
            return "Value must lie between " + NumberRange.Low.ToString() +
                " and " + NumberRange.High.ToString();
        else
            return "Value must NOT lie between " + NumberRange.High.ToString() +
                " and " + NumberRange.Low.ToString();
    }
}
```

The final bit of code for this class is the event handler for the `Validating` event, which changes the background color of the control to yellow if it detects invalid contents. The code for this method is identical to that in the earlier sample, and just as before, you should make sure this method is attached to the `Validating` event.

```
private void NumberTextBoxA_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    if (ContentsInRange)
    {
        this.BackColor = Color.White;
    }
    else
    {
        this.BackColor = Color.Yellow;
    }
    e.Cancel = false;
}
```

This almost completes the code for the `NumberTextBoxA` class, except that we need to add two `using` statements to the top of our class:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.ComponentModel.Design.Serialization;
```

Before we can build the control, we need to make one more minor change. By default, when we created the custom control, it derived from `UserControl`, which contains a property named `AutoScaleMode`. This property is referenced from the default code generated for the `InitializeComponent` method that is in the `NumberTextBoxA.Designer.cs` file:

```
private void InitializeComponent()
{
    components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
}
```

After we change the custom control so that it derives from `System.Windows.Forms.TextBox` instead of `UserControl`, the line of code that does an assignment to `this.AutoScaleMode` will no longer compile. To fix this problem, we just need to remove the assignment, so that the `InitializeComponent` method looks like this:

```
private void InitializeComponent()
{
    components = new System.ComponentModel.Container();
}
```

To modify the `InitializeComponent` method, you can use either of these methods:

- Click the Show All Files button in the toolbar in the Solution Explorer window, and then expand the `NumberTextBoxA` node in the tree view in the Solution Explorer. Then open the `NumberTextBoxA.Designer.cs` file and modify the method.
- Attempt to compile the control. Visual Studio .NET will generate an error because of the offending line. Double-click the error to open the file and find the appropriate line. Delete the line and recompile the control.

Note that this problem will occur whenever you change the base class from `Control` or `UserControl` to another type of control class, such as the `TextBox` class. You will need to fix this problem in the remaining examples in this chapter.

Now you can build the control and add it to the Toolbox if it is not already added.

Creating the test form for this project is virtually identical to the test application we used in the [previous chapter](#) for the `NumberTextBox` project. The only difference is that this time, you need to place `NumberTextBoxA` controls instead of `NumberTextBox` controls on the form. And when you do so, you will find that you can edit your composite property in the Properties window, either with the string or by expanding the property and editing the subproperties.

 Previous

Next 

 Previous

Next 

Debugging Design-Time Code

Parts of the code in the previous example are operational at design-time, including the following:

- Any drawing code of the custom control will execute when the control is drawn in the Design window.
- The constructor of the custom control will execute when the control is built by Visual Studio .NET.
- When any properties are set, the code in the properties executes.

These sections of code actually execute as you are manipulating the custom control in the Design window of Visual Studio .NET and as you are editing properties in the Properties window. You haven't run a program that is using the custom control; you have only placed the control in the Design window.

If you have a bug in your code, perhaps in the `TypeConverter` or in the code to set or get a property value, you can get some very odd behavior from Visual Studio .NET. If you write code that causes an error message to be shown, and dismissing the error message causes the error message to be shown again, or you have a `Paint` event loop, you can get into a situation where you have no option other than to use the Task Manager to kill the Visual Studio .NET process. You then need to restart Visual Studio .NET and fix your code.

However, you can debug your design-time code using Visual Studio itself, by following this procedure:

1. Set breakpoints in the design-time code.
2. If there are multiple projects in a single solution, set the Startup project to the custom control, not the test program for the custom control.
3. Right-click the project for the custom control and select Properties.
4. In the Project Properties dialog box, select the Debug tab. This tab is at the left edge of the Project Properties dialog box, as shown in [Figure 12-5](#).
5. Under Start Actions, select Start External Program, and browse to the Visual Studio executable file, which is `devenv.exe`, as shown in [Figure 12-5](#). This is typically located at `C:\Program`

6. Press F5. This will start another instance of Visual Studio.
7. In this new instance, create a Windows Forms project. Also in the new instance, add the custom control to the Toolbox. Use the Choose Items menu selection on the context menu for the Toolbox.

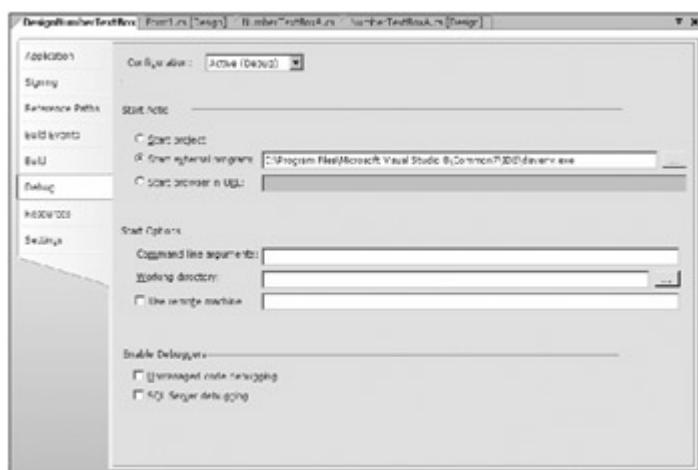


Figure 12-5: Project properties

When you drop your control onto the form in the second instance of Visual Studio, and manipulate it or its properties, your first instance of Visual Studio will hit and stop at your breakpoints. You will then be able to step through your code, examine variables, and so on.

◀ Previous

Next ▶

◀ Previous

Next ▶

Creating a Modal Dialog Box Property Editor

Some properties may not lend themselves to editing via the standard property editor. There are several examples of these properties in the .NET Framework:

- When setting a font for a control, Visual Studio .NET brings up the font selection dialog box. This is a modal dialog that is centered in the screen.
- When setting a `Color` property, Visual Studio .NET brings up a color selection window (a color picker). This window doesn't have any border decorations, such as a close box, the ability to resize the window, and so on. This window is created in the drop-down position.
- When setting the `Anchor` property for a control, Visual Studio .NET brings up a small window that lets you visually indicate where the control is anchored.

We will now make a new version of our `NumberTextBox` custom control that allows us to edit the `NumberRange` in a modal dialog box instead of editing the values in subproperties.

Create a new Windows Control Library project and name it `DesignDlgNumberTextBox`. Then rename `UserControl1.cs` to `NumberTextBoxB.cs`.

In this example, we will need four classes: our custom control, the struct for data validation, the dialog box, and the custom property editor. The struct is identical to the `NumberRange` struct from the previous example. We will retain the name `NumberRange` for this struct, and we won't consider this class further. You can copy this class and the `NumberRangeConverter` class verbatim from the last example. You can also copy the `NumberTextBoxA` class, remembering to rename the class to `NumberTextBoxB`.

Besides these four classes, we will also need a test form to run the application. This form is identical to the previous test forms, apart from containing instances of `NumberTextBoxB` controls instead of `NumberTextBoxA`.

Creating the NumberTextBoxB Control

The NumberTextBoxB class derives from the TextBox class. The code for the class is virtually the same as for the previous example. Since we are using different namespaces, we could theoretically use the same name for this class, but we will instead name it NumberTextBoxB.

Note The sole reason for changing the name of the custom control is to avoid problems if you download the code from the Apress web site. If you downloaded and ran both examples, you might end up with two controls in your Visual Studio .NET Toolbox with the same class name. Despite the different namespaces, this could still be confusing.

The only difference between this custom control and the previous version is the addition of another attribute before the declaration of the NumberRange structure. Modify the attributes as follows:

```
[Editor(typeof(NumberRangeEditor), typeof(UITypeEditor))]
[TypeConverter(typeof(NumberRangeConverter))]
public struct NumberRange
{
    .
    .
    .
}
```

We added an `Editor` attribute, which tells Visual Studio .NET (or any other application that wants to know, for that matter) which class is used to edit instances of this property. This attribute has several overloads, but the one we use takes two parameters, which should both indicate types. The first parameter is the class that Visual Studio .NET can use to control the editing of the property. This class must inherit directly or indirectly from the .NET Framework class `UITypeEditor`. The second parameter is the base class of your class. For most purposes, this parameter seems redundant. It will normally be `UITypeEditor`, but there may be special cases for which you will want to supply a derived class.

Note that `UITypeEditor` is defined in the `System.Drawing.Design` namespace, so we also need a reference to this namespace:

```
using System.Drawing.Design;
```

Next, we need to consider the classes involved with bringing up the dialog box. We will look at the dialog class itself before considering the `NumberRangeEditor` class that actually links the dialog box to the `NumberRange` property.

Adding the NumberRangeDialog

The Windows Forms dialog called `NumberRangeDialog` is the class that represents the dialog box that will be used within Visual Studio .NET to edit `NumberRange` values. The dialog box, when running, looks like [Figure 12-6](#).

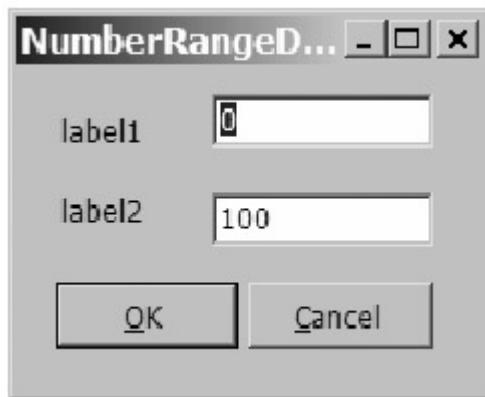


Figure 12-6: The NumberRange dialog box

This dialog box looks quite similar to the actual form that comes up when the program is running, but don't be fooled by the similarity—it is not the same. This dialog box pops up at design-time, not program

runtime, and those are TextBox controls on it, not NumberTextBoxes. (Actually, it would be possible to use NumberTextBoxes, which would leave us in the ironic situation of having a class used to edit instances of itself, but we won't do that here.)

To create this dialog box, create a new Windows Form by right-clicking the project in the Solution Explorer and selecting Add ➤ Add Windows Form. Set the name of the file to NumberRangeDialog.cs, and use the Toolbox to add two TextBox, two Button, and two Label controls, as shown in [Figure 12-6](#).

Now set the following properties for the controls and form. Name one TextBox tbLow and the other tbHigh. Make the buttons standard OK and Cancel buttons, with the Text property of &OK and &Cancel, Name property of btnOK and btnCancel, and DialogResult property of OK and Cancel, respectively.

Select the form by clicking it somewhere, but not on any of the controls, and set the following properties:

- Text: empty
- FormBorderStyle: FixedDialog
- AcceptButton: btnOK
- CancelButton: btnCancel

Now we need to modify the code for the form so that it saves the typed-in values for the NumberRange if the user clicks the OK button (or presses Enter).

Add a private member variable to the NumberRangeDialog class:

```
partial class NumberRangeDialog : Form
{
    private NumberRange editedNR;
```

This field will represent the NumberRange instance that the dialog box is editing.

Add the corresponding public property for this field:

```
public NumberRange EditedNumberRange
{
    get
    {
        return editedNR;
    }
    set
    {
        editedNR = value;
    }
}
```

Modify the constructor of the NumberRangeDialog class to take a NumberRange structure as an argument. This will provide the initial value of the editedNR field:

```
public NumberRangeDialog(NumberRange editedNR)
{
    this.editedNR = editedNR;
    InitializeComponent();
}
```

Double-click the form (but not on a control) to add a Load event handler for the form. Modify the Load event handler as follows:

```
private void NumberRangeDialog_Load(object sender, System.EventArgs e)
{
    tbLow.Text = editedNR.Low.ToString();
    tbHigh.Text = editedNR.High.ToString();
}
```

This Load event handler ensures that the two text boxes are initialized correctly.

Next, add Validating events for the two text boxes:

```
private void tbHigh_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    try
    {
        Double.Parse(tbHigh.Text);
    }
    catch (FormatException)
    {
        MessageBox.Show("Invalid value", "Validation Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        e.Cancel = true;
        return;
    }
}

private void tbLow_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    try
    {
        Double.Parse(tbLow.Text);
    }
    catch (FormatException)
    {
        MessageBox.Show("Invalid value", "Validation Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        e.Cancel = true;
        return;
    }
}
```

These event handlers simply check that the user has typed in numeric values. If he has not, then the handlers display dialog boxes. This isn't a problem—the dialog boxes will be displayed within the Visual Studio .NET environment.

Finally, double-click the OK button to add a Click event that can store the values the user has typed in:

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    editedNR.High = Double.Parse(tbHigh.Text);
    editedNR.Low = Double.Parse(tbLow.Text);
}
```

This code ensures that when the user selects OK in the dialog box, the new values for the NumberRange are stored and can be later accessed using the NumberRangeDialog.EditedNumberRange property.

As explained next, the NumberRangeEditor class will retrieve the value of this property so that it can inform Visual Studio .NET of the new value.

Coding the NumberRangeEditor for NumberTextBoxB

The class for our custom property editor, NumberRangeEditor, class derives from UITypeEditor, and is a class that tells Visual Studio .NET how to bring up the NumberRangeDialog dialog box to edit the property. We told Visual Studio .NET about this class through the Editor attribute.

Before we examine the code for this class, let's examine in principle what Visual Studio .NET is expecting our code to do. In order to instantiate the dialog box, the IDE invokes a number of virtual methods defined in the UITypeEditor class. Visual Studio .NET never really communicates directly with our NumberRangeDialog class. It doesn't know how to, because the definition of that class is entirely up to us. Instead, Visual Studio .NET talks to our UITypeEditor-derived class,

`NumberRangeEditor`, and asks it to arrange everything. So, our `NumberRangeEditor` must be able to talk to Visual Studio .NET on the one hand, which it does by implementing certain `UITypeEditor` methods that the IDE knows about, and to our dialog box on the other hand, which it clearly does in the implementations of those methods that we will write.

We need to override the following methods:

- `EditValue()` : This method is called when the developer clicks the button to call up the editor for a property. It is passed the current value of the property and must display the dialog box, obtain any new values from the dialog box, and return the new value of the property, cast to object.
- `GetEditorEditStyle()` : This method returns an enum value indicating the style of the dialog box (or drop-down list) that is to be used.

When we drop a control onto the Design window in Visual Studio .NET, the IDE creates an instance of our custom control. This object draws itself in the Design window. It also contains the values of the properties that we set in the Properties window. In addition, when we build a custom property editor, this instance of our custom control is passed as an argument to the `EditValue()` method, in the `value` parameter. One of the first things that this method does is create a variable of the type of our custom control and assign the `value` parameter (while casting) to the object. We will need this custom control for a variety of reasons.

So, we need to create a class that derives from `UITypeEditor`. This class is a simple C# class, not a control, so you can just create a new class in the project, call it `NumberRangeEditor.cs`, and then edit the base class. We will also need a few namespace references:

```
using System;
using System.ComponentModel;
using System.Drawing.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using System.Globalization;
using System.ComponentModel.Design.Serialization;

namespace DesignDlgNumberTextBox
{
    ...
    // Remember to put this class at the end of the namespace
    public class NumberRangeEditor : System.Drawing.Design.UITypeEditor
    {
```

First, we will define the `UITypeEditorStyle()` method:

```
public override UITypeEditorEditStyle GetEditStyle(
    ITypeDescriptorContext context)
{
    if (context != null && context.Instance != null)
    {
        return UITypeEditorEditStyle.Modal;
    }
    return base.GetEditStyle(context);
}
```

One other method that we must implement in this class is the `GetEditStyle()` method. There are two styles of property editors. The style that we are building right now is a modal dialog box style. All this method does is return `UITypeEditorEditStyle.Modal`. In the next example, where we implement a drop-down style of property editor, we will change this method appropriately. This method gets passed an interface reference to the interface `System.ComponentModel.ITypeDescriptorContext`. We won't worry about this interface here, since, beyond checking that it isn't null, we don't need it.

Note If your method requires it, you can use the

`System.ComponentModel.ITypeDescriptorContext` interface to obtain information about the property that you are editing. You might do this, for example, if you were using the same editor class to control the editing of different properties of different types.

Now let's check out that `EditValue()` method:

```
public override object EditValue(
    ITypeDescriptorContext context, IServiceProvider provider, object value)
{
    IWindowsFormsEditorService edSvc = null;

    if (context != null
        && context.Instance != null
        && provider != null)
    {
        edSvc = (IWindowsFormsEditorService)
            provider.GetService(typeof(IWindowsFormsEditorService));

        if (edSvc != null)
        {
            NumberTextBoxB originalControl = (NumberTextBoxB)context.Instance;
            NumberRangeDialog editingDialog =
                new NumberRangeDialog (originalControl.NumberRange);

            DialogResult dr = edSvc.ShowDialog(editingDialog);

            // This method could take different actions based on
            // the return value of the dialog
            if (dr == DialogResult.OK)
            {
                value = editingDialog.EditedNumberRange;
                return value;
            }
        }
    }
    return value;
}
```

The `EditValue()` method also takes an `ITypeDescriptorContext` as a parameter, and this time, we are going to need to use it. This method takes a second parameter, which is a reference to a `System.IServiceProvider` interface. `IServiceProvider` defines one method, `GetService()`, which returns a reference to the object inside Visual Studio .NET that should call the dialog box, cast to object, as well as a third parameter that takes the return value of the property, if any.

Let's look at that second parameter in more detail. Since we are building a property editor, Visual Studio .NET needs to do the actual opening of the dialog box. From the `IServiceProvider` argument, we can get an instance of a class that implements the `IWindowsFormsEditorService` interface. This interface defines a method, `ShowDialog()`, which we call when we want to put up our dialog box. That way, it is still Visual Studio .NET that controls the actual showing of the dialog box. It sounds more complicated than it actually is. We just follow the recipe, as seen in this example:

```
edSvc = (IWindowsFormsEditorService)
    provider.GetService(typeof(IWindowsFormsEditorService));
```

Followed later by this:

```
DialogResult dr = edSvc.ShowDialog(editingDialog);
```

We use the `context` argument to obtain the original value of the `NumberRange` property, so we can pass it to the dialog box:

```
NumberTextBoxB originalControl = (NumberTextBoxB)context.Instance;
NumberRangeDialog editingDialog =
    new NumberRangeDialog (originalControl.NumberRange);

DialogResult dr = edSvc.ShowDialog(editingDialog);
```

Once the call to `ShowDialog()` has returned, we simply return the result if necessary.

We are finally finished with our custom control. Compile it and add it to the Toolbox. Now we need to

Note Before building the control, modify the `InitializeComponent` method for the control to remove the assignment to `this.AutoScaleMode`, as explained in the "[Creating the NumberTextBoxA Control](#)" section earlier in this chapter.

Testing the NumberTextBoxB Control

Add a new Windows Application project to the solution and name it `TestDesignDlgNumberTextBox`. Set it as the startup project and drag two `NumberTextBoxB` controls onto the form.

Now, you can edit the `NumberRange` property in three ways:

- Edit it as a comma-delimited string.
- Expand the property and edit subproperties.
- Click the button to the right of the value of the property, and edit the parameters in the dialog box.

This is a very simple example, but you could build a much more elaborate dialog box for editing more complex data types.

 Previous

Next 

 Previous

Next 

Creating a Drop-Down Property Editor

A different style of property editor is one that "drops down" below the property value. This type of property editor provides a sense of context to the developers using the custom control. Since the window in which they edit the property is directly below the property, there is a visual link between the two. In addition, the user interface is slightly different from that of a dialog box. A drop-down property editor is a window that automatically closes when the developer using the control clicks outside the property-editing window.

To create this type of property editor, we need to create a custom control to use in the `UITypeEditor`, rather than creating a dialog box, as we did in the previous example to present a modal dialog box property editor.

Just as in the previous example, we need four classes in our DLL project, plus a Windows Form class to test the application. Most of these classes are, however, very similar to the previous project, and we will give some of them the same class name, distinguished by the namespace

`DesignDropDownNumberTextBox` for this example. Here are the classes we need in our project:

- The class representing the number text box has identical code to the previous example, but this time we will name the class `NumberTextBoxC`, so its name will be different in the Toolbox.
- Our custom property editor derived from `UITypeEditor` is similar to the `NumberRangeEditor` from the previous sample and will have the same name.
- The `NumberRange` class used for data validation remains identical to the previous example.
- The `NumberRangeDropDown` is the drop-down control. This class is new and replaces the `NumberRangeDialog` class from the previous example.
- The main form class we use to test the control, `Form1`, is identical to the previous example, except that it contains two `NumberTextBoxC` controls in place of the `NumberTextBoxB` ones.

In this section, we'll look at the classes that are different for the drop-down property editor.

As usual, begin by creating a new Windows Control Library project, and name the solution and the DLL project `DesignDropDownNumberTextBox`.

Creating the NumberRangeDropDown Control

The NumberRangeDropDown custom control will be the custom control in which we will edit the NumberRange structure. This custom control will be identical in nature to the composite custom control that we built in the [previous chapter](#). It will have four constituent controls.

On the menu, select Project ➤ Add User Control and set the name of the control to NumberRangeDropDown.cs. Add two Label controls and two TextBox controls, and lay them out as shown in [Figure 12-7](#).



Figure 12-7: Layout of the drop-down custom control

Give one TextBox control a Name property of tbLow, and the other tbHigh. Then select the NumberRangeDropDown custom control (by clicking the Design window somewhere outside any of the controls) and set the BackColor property to ControlLight.

Now we need to edit the code. Bring up the code editor for the NumberRangeDropDown custom control and add the following private member variables to the NumberRangeDropDown class:

```
private NumberRange originalNR;
private NumberRange newNR;
private bool canceling;
```

These variables will store, respectively, the old value of the NumberRange, the new value the user has typed in, and whether the user has exited the control in a manner that requires the new value not to be saved (in other words, by pressing the Esc key).

Add the following public property to the NumberRangeDropDown class, which allows access to the new value of the NumberRange:

```
public NumberRange NewNumberRange
{
    get
    {
        return newNR;
    }
}
```

Modify the constructor of the NumberRangeDropDown class to take a NumberRange structure as an argument:

```
public NumberRangeDropDown(NumberRange originalNR)
{
    this.originalNR = originalNR;
    InitializeComponent();
}
```

Double-click the Design window for the custom control (but not on one of the constituent controls) to add a Load event handler for the custom control. This event will need to initialize the two text boxes with the old NumberRange value. So, modify the Load event handler as follows:

```
private void NumberRangeDropDown_Load(object sender, System.EventArgs e)
{
    // Save a copy of NumberRange, so that if the user presses the
    // Esc key, we can revert to the previous values
    newNR = originalNR;

    // Load up the TextBox controls
    this.tbHigh.Text = originalNR.High.ToString();
    this.tbLow.Text = originalNR.Low.ToString();
}
```

Add a Validating event handler for the tbLow TextBox and ensure it is hooked up to the Validating event for tbLow:

```
private void tbLow_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    try
    {
        Double.Parse(tbLow.Text);
    }
    catch (FormatException)
    {
        MessageBox.Show("Invalid value", "Validation Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        e.Cancel = true;
        return;
    }
}
```

Do the same thing for tbHigh:

```
private void tbHigh_Validating(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    try
    {
        Double.Parse(tbHigh.Text);
    }
    catch (FormatException)
    {
        MessageBox.Show("Invalid value", "Validation Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        e.Cancel = true;
        return;
    }
}
```

Now we need to consider what happens when the control closes. Since this is a drop-down parameter editor, it will automatically close in three situations:

- If the developer presses the Enter key
- If the developer presses the Esc key
- If the developer clicks outside the custom control

The behavior that we want is that if the developer using NumberRangeDropDown to enter validation parameters presses the Enter key or clicks outside the custom control window, the changed parameter values are committed. However, if the developer presses the Esc key, any changes to the parameter values should be discarded. In order to implement this, we need to do the following:

- Save the previous values in the Load event for the NumberRangeDropDown custom control.
- Override the ProcessDialogKeys() virtual method in the NumberRangeDropDown custom control. If this method gets an event for the Esc key, it needs to set a flag indicating that the user pressed the Esc key.
- In the Leave event, set the NumberRange property, unless the user pressed the Esc key.

Implement the ProcessDialogKey() method like this:

```
protected override bool ProcessDialogKey(Keys keyData)
{
    if (keyData == Keys.Escape)
    {
        originalNR = newNR;
        canceling = true;
    }
}
```

```

        }
        return base.ProcessDialogKey(keyData);
    }
}

```

Add an event handler for the `Leave` event, and modify it like this:

```

private void NumberRangeDropDown_LostFocus(
    object sender, System.EventArgs e)
{
    if (! canceling)
    {
        newNR.Low = Double.Parse(tbLow.Text);
        newNR.High = Double.Parse(tbHigh.Text);
    }
}

```

The `NumberRangeDropDown` control is now finished. All we need to do is add the code to tell Visual Studio .NET that this control is the editor for our `NumberRange` property.

Coding the NumberRangeEditor for NumberTextBoxC

As in the previous example, the `NumberRangeEditor` is a plain C# class derived from `UITypeEditor`. The code for it is fairly different from the previous version, so we will go through it in detail (highlighting only those lines of code that are not the same). We still need to override the same methods, `EditValue()` and `GetEditStyle()`, but our implementations are somewhat different.

First, add the required namespace references and class declaration:

```

using System;
using System.ComponentModel;
using System.Drawing.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using System.Drawing.Design;

using System.Globalization;
using System.ComponentModel.Design.Serialization;

namespace DesignDropDownNumberTextBox
{
    // Remember to put this class at the end of the namespace
    /// <summary>
    /// Summary description for NumberRangeEditor.
    /// </summary>
    public class NumberRangeEditor : System.Drawing.Design.UITypeEditor
    {
}

```

Next, let's examine the `GetEditStyle()` method:

```

public override
    UITypeEditorEditStyle GetEditStyle(ITypeDescriptorContext context)
{
    if (context != null && context.Instance != null)
    {
        return UITypeEditorEditStyle.DropDown;
    }
    return base.GetEditStyle(context);
}

```

Here, we return a value to indicate that the control is a `DropDown` control. Now, let's look at the changes for the `EditValue()` method:

```

public override object EditValue(
    ITypeDescriptorContext context, IServiceProvider provider, object value)
{
    IWindowsFormsEditorService edSvc = null;
}

```

```
if (context != null  
    && context.Instance != null  
    && provider != null)  
{  
    edSvc = (IWindowsFormsEditorService)provider.GetService(  
        typeof(IWindowsFormsEditorService));  
  
    if (edSvc != null)  
    {  
        NumberTextBoxC originalControl = (NumberTextBoxC)context.Instance;  
        NumberRangeDropDown editingDropDown =  
            new NumberRangeDropDown  
            (originalControl.NumberRange);  
        edSvc.DropDownControl(editingDropDown);  
        value = editingDropDown.NewNumberRange;  
        return value;  
    }  
}  
return value;  
}
```

As you can see from this code, getting a reference to the instance of our NumberTextBoxC custom control is identical to the previous version of NumberRangeEditor. However, instead of creating an instance of the dialog box, we create an instance to the drop-down custom control.

The `EditValue()` method of `NumberTextBoxParametersEditor` uses a different technique from the previous example to put up the custom property editor. In the previous version, the `EditValue()` method used the `ShowDialog()` method. This version uses the `IWindowsFormsEditorService.DropDownControl()` method:

```
edSvc.DropDownControl(editingDropDown);
```

Whereas in the previous version, the dialog box was able to return a value, either `DialogResult.OK` or `DialogResult.Cancel`, in this version, we cannot get a return value from the `NumberRangeDropDown` custom control. However, we have a public property in the `NumberRangeDropDown` custom control that contains the edited values:

```
value = editingDropDown.NewNumberRange;  
return value;
```

We coded the `NumberRangeDropDown` custom control so that if the developer using the control presses the Esc key, the `NumberRange` property is not modified. Therefore, we can simply set the `value` object to the `NewNumberRange` property of the `NumberRangeDropDown` custom control. The behavior will be as expected.

We are finished with our custom control. Compile it and add it to the Toolbox, remembering to remove previous versions of the control from the Toolbox window to avoid any conflict.

Note Before building the control, modify the `InitializeComponent` method for the control to remove the assignment to `this.AutoScaleMode`, as explained in the "[Creating the NumberTextBoxA Control](#)" section earlier in this chapter.

Testing the NumberTextBoxC Control

Now build a test application, using the same test form as the previous example, except using `NumberTextBoxC` controls instead of `NumberTextBoxB` controls. Bring up the Properties window and click the down arrow that appears when you click the `NumberRange` property. The drop-down editor should look like [Figure 12-8](#).

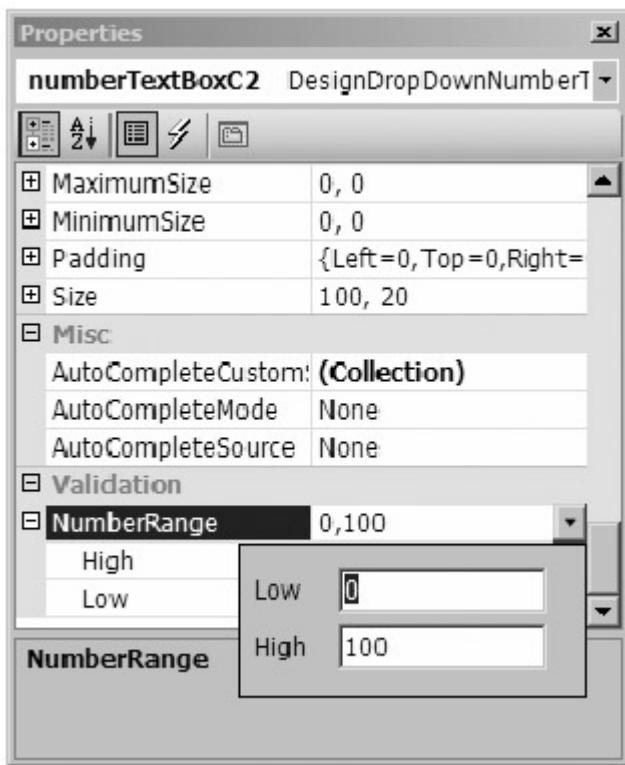


Figure 12-8: Drop-down number range editor

◀ Previous

Next ▶

◀ Previous

Next ▶

Implementing a Custom Designer

Sometimes, editing properties is not the most convenient way to modify the configuration of a custom control. For example, if you were making an elaborate spreadsheet custom control, it certainly would be possible to enter the widths of the columns into the Properties window. However, this would be cumbersome at best and error-prone at worst. It would be far more convenient to allow the developer using the control to directly manipulate the column widths using the mouse. To get this functionality, you can implement a custom designer—a class that can give Visual Studio .NET the information it needs to directly draw your control in the Design window for you to edit it.

There is a range of functionality that you can implement using a custom designer. Some of the most important functionality is as follows:

- Draw adornments on top of the control. The routine that draws adornments gets the chance to do its drawing after the control has drawn itself.
- Set the sizing and movement behavior of the control. You can choose whether the control can be sized by its top, left, bottom, or right side. You can specify that the control may or may not be moved.
- Write code to set the default values for any property. For example, you might use this if default values for one control depend on property values in another control.
- Write code to respond to mouse-dragging within the control. A spreadsheet control that allows the developer to directly manipulate column widths would be a good example of the use of this feature.
- Add items to the context menu that developers using the custom control see when they right-click the control.
- If your custom control is a composite control, allowing other controls to be dropped on it, you can

manipulate and query the constituent controls.

To give you an idea about the capabilities of custom designers and how to implement them, we will develop two examples. The first, `SimpleBlankControl`, is about as simple a control as you can get—all it does is paint its backdrop yellow. We will attach this to a designer that causes the Visual Studio .NET Design view to add a menu item to its context menu for the control, allowing you to set the control's size to make it square. The second example is a bit more complex and features a control that can draw a vertical line. The corresponding designer allows you to move the line in the Visual Studio .NET Design view by dragging it.

Creating the SimpleBlankControl

Our custom control, `SimpleBlankControl`, is very simple. However, it clearly demonstrates the use of a custom designer. You can edit the control in the Visual Studio .NET Design view; for example, changing its size by dragging with the mouse.

Building the Control

Create the project in the usual way and make sure the control is called `SimpleBlankControl` and derived from `Control`. The only change you need to make to `SimpleBlankControl` is to use the Properties window to change its `BackColor` property to Yellow, and then modify the code for the class declaration by adding an attribute:

```
namespace SimpleBlankControl
{
    [Designer(typeof(SimpleBlankControlDesigner))]
    public partial class SimpleBlankControl : System.Windows.Forms.Control
    {
```

The `Designer` attribute indicates which class can be used as a custom designer. Here, it indicates a class that we are about to code, `SimpleBlankControlDesigner`.

Building the Designer

The `SimpleBlankControlDesigner` class derives from the `ControlDesigner` class. This particular version simply adds a menu item to the context menu. Since the `SimpleBlankControl` class is derived from `Control`, Visual Studio .NET will already supply the ability to draw the control in the Design view and resize it.

Add a new class named `SimpleBlankControlDesigner` to the project using Project ➤ Add Class, and then add the following using statements:

```
using System;
using System.Design;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
```

We must also add a reference to the `System.Design` assembly to our project in order to access the `ControlDesigner` class from which our class is derived. The complete definition of the class looks like this:

```
public class SimpleBlankControlDesigner :
    System.Windows.Forms.Design.ControlDesigner
{
    public override DesignerVerbCollection Verbs
    {
        get
        {
            DesignerVerb[] verbs = new DesignerVerb[]
            {
                new DesignerVerb("&Make Square", new EventHandler(OnMakeSquare))
            };
            return new DesignerVerbCollection(verbs);
        }
    }
}
```

```
private void OnMakeSquare(object sender, EventArgs e)
{
    Size s = this.Control.Size;
    s.Width = Math.Max(s.Width, s.Height);
    s.Height = Math.Max(s.Width, s.Height);
    PropertyDescriptorCollection properties =
        TypeDescriptor.GetProperties(this.Control.GetType());
    PropertyDescriptor sizeProp = properties["Size"];
    sizeProp.SetValue(this.Control, s);
}
```

As you can see, this class has one property and one method.

The `Verbs` property does the actual adding of the menu item, and it is overridden from `ControlDesigner`. Visual Studio .NET knows about this method and uses it to find out about any extra items (verbs) that need to be added to the context menu when the mouse is over the control in the Design view. The property is of type `DesignerVerbCollection`, a class that represents a set of `DesignerVerb` instances. A `DesignerVerb` object contains a string that indicates the text to be displayed in the context menu and an event handler that is invoked when the developer clicks that context menu item.

The `OnMakeSquare()` method is the event handler that we supply to the Make Square context menu item. It identifies the larger dimension of the control (width or height) and sets both dimensions to this value, making the shape of the control square. Note that to change a property of our control, we use a `PropertyDescriptor`. If you changed the property directly, the Design window may not properly redraw itself.

Testing the Project

As usual, we compile the control and add it to the Toolbox. Create a test Windows Forms application and drag a `SimpleBlankControl` instance onto it. You will then find that if you right-click the control in the Design view, the Make Square menu option will appear, as shown in [Figure 12-9](#).

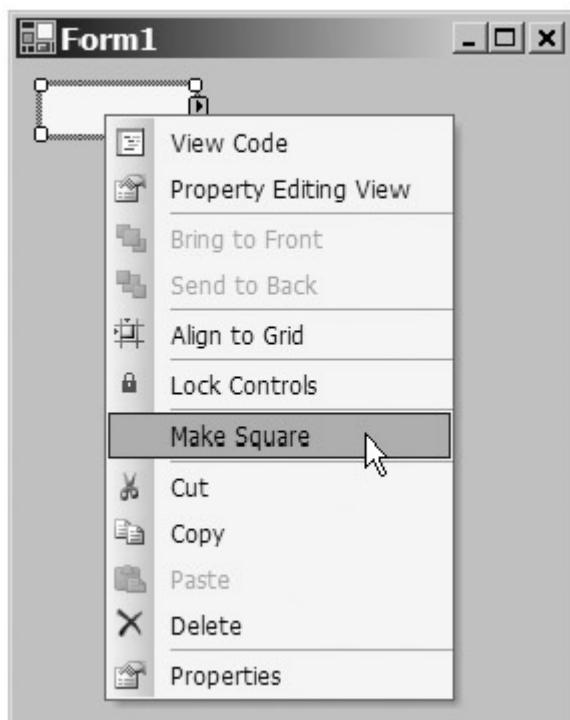


Figure 12-9: A customized context menu

Creating the SimpleLineControl

Our next version of a custom designer will be somewhat more complicated. First, we will make a custom control that draws a vertical line at the position of a property, as shown in [Figure 12-10](#).

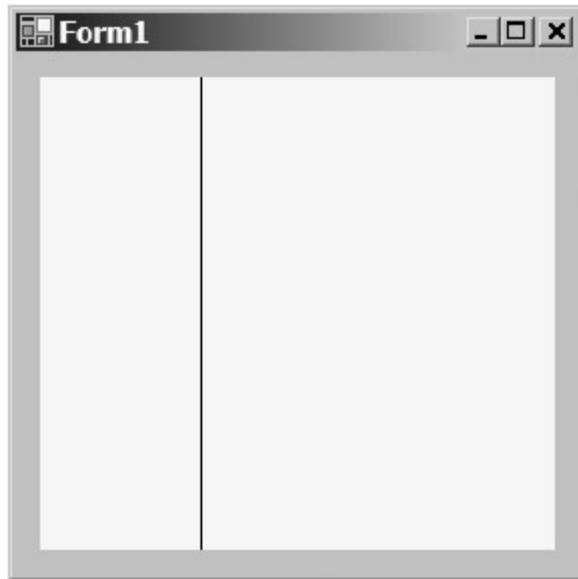


Figure 12-10: The SimpleLineControl

We will create a designer for this control, which will add an adornment—a small rectangle. We will allow the developer using the control to click and drag on the adornment. As the developer drags on the adornment, we will change the value of the property that controls where the line is, so that at runtime, the line will be drawn at the position where the developer places it in the Design view. When you look at this custom control in the designer, there is a small rectangle on the line, as shown in [Figure 12-11](#).

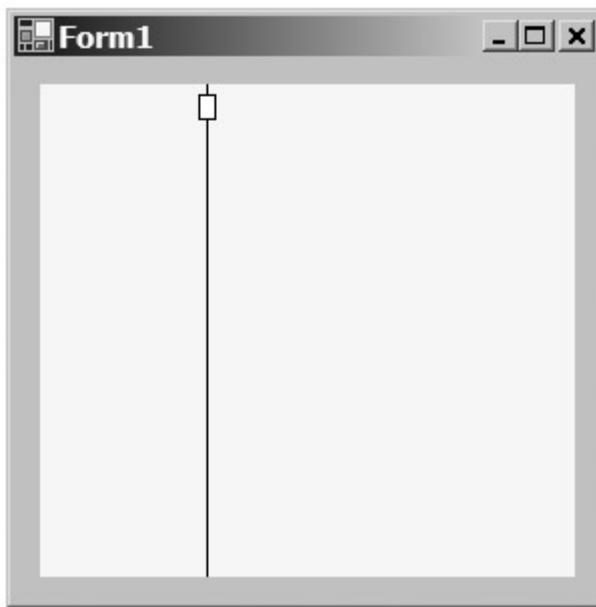


Figure 12-11: The adornment in the Design window

You can move your mouse into that rectangle, and then drag. When dragging, the code changes the `LinePosition` property of the `DragLine` control, and then causes it to redraw itself.

Building the SimpleLineControl

Create a new project with a custom control, this time named `SimpleLineControl`. Then add the `Designer` attribute to the control definition:

```
namespace SimpleLineControl  
{
```

```
[Designer(typeof(SimpleLineControlDesigner))]  
public class SimpleLineControl : System.Windows.Forms.Control  
{
```

Change the BackColor property of the control to Yellow and add the following field and property to indicate the position of the vertical line:

```
private int linePosition = 5;
```

```
[Category("Appearance")]  
[DefaultValue(5)]  
public int LinePosition  
{  
    get  
    {  
        return linePosition;  
    }  
    set  
    {  
        linePosition = value;  
    }  
}
```

Then add a Paint event handler to the control and modify the code as follows:

```
private void SimpleLineControl_Paint(  
    object sender, System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.DrawLine(Pens.Black, LinePosition, 0,  
              LinePosition, ClientRectangle.Height);  
}
```

Building the SimpleLineControlDesigner

In this custom designer, we will add code to draw the small rectangle adornment and to respond when the user drags it. We create the code by adding a simple C# class to the project, called SimpleLineControlDesigner.cs. We then add the following namespace references:

```
using System;  
using System.Design;  
using System.ComponentModel;  
using System.ComponentModel.Design;  
using System.Drawing;  
using System.Windows.Forms;
```

Next, we define the class and add a couple of member fields and some constants:

```
namespace SimpleLineControl  
{  
    public partial class SimpleLineControlDesigner :  
        System.Windows.Forms.Design.ControlDesigner  
    {  
        private bool dragging = false;  
        private int lastXPos = -1;  
        bool mouseInAdornment = false;  
  
        private const int WM_MOUSEMOVE      = 0x0200;  
        private const int WM_LBUTTONDOWN    = 0x0201;  
        private const int WM_LBUTTONUP     = 0x0202;  
        private const int WM_LBUTTONDBLCLK = 0x0203;  
        private const int WM_RBUTTONDOWN    = 0x0204;  
        private const int WM_RBUTTONUP     = 0x0205;  
        private const int WM_RBUTTONDBLCLK = 0x0206;
```

The dragging field indicates whether we are currently in a drag operation, lastXPos indicates the

horizontal position of the line while being dragged, and `mouseInAdornment` indicates if the mouse is currently inside the adornment. All of these fields will be updated as the user moves the mouse, as you will see soon.

The constants that we define are the values of certain low-level Windows messages. Due to the way the control designer architecture has been designed, we won't be able to use the standard Windows Forms events to handle mouse movements and clicks in the Design view. Instead, we must use a traditional-style message-handling method, known as a *windows procedure*, or `WndProc`, which is what is happening behind the scenes in Windows Forms events. (C++ and ex-C++ developers will no doubt recognize the message names, but they may not be familiar to VB and ex-VB developers.)

Basically, a message-handling routine (equivalent to an event handler) is called, but instead of there being a different handler for each event, there is one routine, and it is passed a number (known as a *message*) that indicates what has happened. Thus, for example, if the user has moved the mouse, it gets passed the number 0x0200 (512 in decimal), this value having been traditionally given the name `WM_MOUSEMOVE`.

Next, we override the `ControlDesigner.OnPaintAdornments()` method, which Visual Studio .NET calls after painting the control to paint any adornments. We also have a helper method, `calcAdornmentRect()`, which figures out where to draw the rectangle. The rectangle should be drawn four pixels to the left of the line position and five pixels from the top of the control, and it should have the size (8, 12):

```
private Rectangle calcAdornmentRect()
{
    int XPos = ((SimpleLineControl)(this.Control)).LinePosition;
    return new Rectangle(XPos - 4, 5, 8, 12);
}

protected override void OnPaintAdornments(PaintEventArgs pe)
{
    Graphics g = pe.Graphics;
    Rectangle r = calcAdornmentRect();
    g.FillRectangle(Brushes.White, r);
    g.DrawRectangle(Pens.Black, r);
}
```

Note that we obtain the actual `SimpleLineControl` being painted as the `ControlDesigner.Control` property.

We also override the `ControlDesigner.OnSetCursor()` method that is responsible for changing the mouse cursor. The semantics that we must implement are a bit strange. If we have changed the mouse cursor, this method needs not to call up to the base class. If we haven't changed the mouse cursor, then we must call up to the base class. As you will see when we examine the `WndProc()` method, we always set the cursor ourselves if the mouse is over the adornment rectangle or the developer is dragging the line. Our `OnSetCursor()` override looks like this:

```
protected override void OnSetCursor()
{
    if (mouseInAdornment == false || !dragging)
        base.OnSetCursor();
}
```

Now comes the final method in our class, and it is a big one! It is overridden from `ControlDesigner`, and it is the Windows procedure that is called in lieu of mouse events. Its parameter is a `Message` struct, whose `Msg` property gives the message value that indicates why the method has been called. Like traditional Windows procedures, the structure of this method is basically a big `switch` statement depending on the message passed to it:

```
protected override void WndProc(ref Message m)
{
    Point p;
    Rectangle r;
    SimpleLineControl dl;

    switch (m.Msg)
    {
```

```
case WM_LBUTTONDOWN:
    r = calcAdornmentRect();
    p = new Point(m.LParam.ToInt32());
    if (r.Contains(p))
    {
        dragging = true;
        this.Control.Parent.Cursor = Cursors.Arrow;
    }
    else
    {
        base.WndProc(ref m);
    }
    break;

case WM_MOUSEMOVE:
    p = new Point(m.LParam.ToInt32());
    dl = (SimpleLineControl)(this.Control);
    if (dragging)
    {
        int XPos = p.X;
        XPos = Math.Max(5, XPos);
        XPos = Math.Min(dl.ClientRectangle.Width - 5, XPos);
        if (lastXPos != XPos)
        {
            PropertyDescriptorCollection properties =
                TypeDescriptor.GetProperties(this.Control.GetType());
            PropertyDescriptor sizeProp = properties["LinePosition"];
            sizeProp.SetValue(this.Control, XPos);
            lastXPos = XPos;
            dl.Invalidate();
            dl.Update();
        }
        this.Control.Parent.Cursor = Cursors.Arrow;
        return;
    }

else
{
    r = calcAdornmentRect();
    p = new Point(m.LParam.ToInt32());
    if (r.Contains(p))
    {
        mouseInAdornment = true;
        this.Control.Parent.Cursor = Cursors.Arrow;
    }
    else
    {
        mouseInAdornment = false;
    }
    base.WndProc(ref m);
}
break;

case WM_LBUTTONUP:
if (dragging)
{
    dl = (SimpleLineControl)(this.Control);
    p = new Point(m.LParam.ToInt32());
    int XPos = p.X;
    XPos = Math.Max(5, XPos);
    XPos = Math.Min(dl.ClientRectangle.Width - 5, XPos);
    PropertyDescriptorCollection properties =
        TypeDescriptor.GetProperties(this.Control.GetType());
    PropertyDescriptor sizeProp = properties["LinePosition"];
}
```

```
    sizeProp.SetValue(this.Control, XPos);
    lastXPos = -1;
    dl.Invalidate();
    dl.Update();
    dragging = false;
}
base.WndProc(ref m);
break;

default:
base.WndProc(ref m);
break;
}
}
```

This code basically checks what message has been passed to it, and then does the following:

- If it has been called because the user has clicked the left mouse button down, check if the mouse is in the adornment rectangle. If so, record that a dragging operation has just started; otherwise, pass control to the base class.
- If the user has moved the mouse, check whether we are in the middle of a dragging operation. If so, we need to redraw the control with the vertical line and adornment in the new dragged position. Otherwise, we don't need to do anything besides make a note of whether the mouse is in the adornment rectangle.
- If this method has been called because the left mouse button has been released, we need to check if we were in a dragging operation. If so, then we need to record that the drag operation has finished, and update the `LinePosition` property of the control we are editing so that it contains the new location of the vertical line.

The `WndProc()` method needs to set a state variable (the `dragging` variable). The mouse-down event sets it, and the mouse-move and mouse-up events use the state of the `dragging` variable. This is a common method for programming dragging operations at the lowest level.

When the method changes the value of `LinePosition`, it calls the `Invalidate()` and the `Update()` methods on the custom control, which causes the control to redraw itself:

```
dl.Invalidate();
dl.Update();
```

This `WndProc()` function can process any number of messages, but it will not, of course, process every message that gets passed to it. Any messages that our function doesn't process needs to be passed to the base class.

Note A real-life control with a custom editor—such as a spreadsheet control that lets the user directly manipulate column widths and move columns around—would be hundreds of lines of code. However, the implementation of such a complex control is basically the same as the simple example presented here.

That completes the control. To build the example, you will need to add a reference to the `System.Design.dll` assembly. To try it out, create a test Windows Forms application and place a `SimpleLineControl` control on the form.

 Previous

Next 

 Previous

Next 

Summary

In this chapter, you saw how you can enhance the design-time experience of developers using custom controls. The examples demonstrated how it's possible to significantly change the way the developer using your controls interacts with them in Visual Studio .NET.

A very important and relatively simple method of enhancing design-time support is to create a

TypeConverter for your custom control. A TypeConverter provides three fundamental areas of functionality:

- It converts to and from strings, so that a data structure such as a structure or class can be displayed as a string in the Properties window. If feasible, the developer can also set the values for a structure or class by entering a string.
- It provides information about subproperties, so that the developer can expand the property in the Properties window and edit the subproperties.
- It provides information to Visual Studio .NET, so that the code generator can properly generate the code to persist the values in the properties of the control.

We also covered creation of a custom user interface for editing properties. There are two styles. With the first style, the developer using the control edits the properties in a modal dialog box. With the second style, the developer using the control edits the properties in another custom control that appears in the drop-down position below the property being edited.

For very complex custom controls, or those for which you want to provide more direct manipulation of the custom control, you can implement a custom designer. With a custom designer, you can allow the developer using the control to directly manipulate it using the mouse. You also might provide more information about the control, so that the developer doesn't need to look at the Properties window to get a quick overview of all of the controls in a window.

With previous custom control technologies, much of the design-time functionality was embedded in the design tool. With Visual Studio .NET and the .NET Framework, *all* design-time functionality is encapsulated by the custom control itself. This gives you unprecedented power to develop much more usable custom controls.

 Previous

Next 

 PreviousNext 

Chapter 13: Scrolling

Overview

When you're building a custom control, you may encounter the problem of too much data and too little screen real estate. You can take a couple of approaches to solve this problem. For example, you could create a tabbed user interface that allows the user to switch between views, or create subsidiary or child windows based on user interactions. A common approach, and the one addressed in this chapter, is to create a scrolling region, just as you find in word processors, where you scroll down or across to see the entire document.

Sometimes the entire document is called a *virtual space*, and the "window" into the document is called the *visible space*. The visible space might occupy an entire window, or it may occupy just a portion of a window. In either case, you can consider the visible space as a *viewport* into the larger virtual space. As an example of this, when you are looking at your code in the code editor in Visual Studio .NET, the virtual space contains the entire code listing for the source code file you are examining. It is very rare that your code is short enough to fit in the code window, so the window has scroll bars to allow you to scroll through the code.

There is considerable built-in support for scrolling in the Windows Forms classes, but there are also situations in which you may prefer to handle scrolling yourself. In this chapter, we will look at what Windows Forms can do for you, and then examine some specific situations in which you might want to handle scrolling from your code, instead of relying on the .NET Framework class library code.

You might have expected this topic to have been discussed earlier in this book, when we were focusing more on GDI+. In previous implementations of graphical programming toolkits, this would be the case. However, when using Windows Forms, the easiest and best way to put together a scrolling region is to implement it as a custom control. It is possible to build a scrolling region without first building a custom control, but it is much more cumbersome. For this reason, I delayed delving into this topic until after the coverage of building custom controls. If you are not familiar with the process of building custom controls, read [Chapters 11](#) and [12](#) before continuing with this chapter. This chapter builds on the information presented in those chapters.

 PreviousNext  PreviousNext 

Simple Scrolling

Let's start with an extremely simple example that demonstrates building a control capable of scrolling, using just the built-in support. It will simply be a form with a control that displays an image.

For this example, create a Windows Forms application and call it `ScrollingPicture`. We won't even derive our own custom control; instead, we will make use of the `Panel` class. So, drag a Panel control from the Toolbox onto the form. Then set the background image to some image file on your computer. Any file will do, as long as it's a reasonably large image—big enough that it won't fit completely into the panel on your form. At this point, running the sample should show something like [Figure 13-1](#) (using a JPG file of the side of a car as the background image).



Figure 13-1: Picture in a Panel control

Since the image is too big to fit in the panel, you have no way of seeing anything other than the top-left corner of it. Obviously, you need scroll bars.

Adding Scroll Bars

To add scroll bars, you need to set two properties on the panel: `AutoScroll` and `AutoScrollMinSize`.

`AutoScroll` is a Boolean property that indicates whether the control should be prepared with the ability to display scroll bars. So, locate this property in the Properties window and set it to `True` for the Panel control.

`AutoScrollMinSize` is of type `Size`. It stores the size of the virtual space, meaning the total size you would ideally have liked the control to have been if your screen were big enough. Clearly, there is no point in displaying scroll bars unless this size is bigger than the actual size of the control, and that is exactly what happens. The control will display a vertical scroll bar only if `AutoScrollMinSize.Height` is greater than its `Size.Height` property, and it will display a horizontal scroll bar only if `AutoScrollMinSize.Width` is greater than its `Size.Width` property. `AutoScrollMinSize` is also used by the control when calculating how much to allow the user to scroll and how large to make the scroll bars.

You will find `AutoScrollMinSize` in the Properties window, too, and you can set it there, but we won't for this example. Instead of setting a hard-coded value for it, we will have the control calculate the value at runtime from the size of the image it is displaying. So, make the following change to the constructor of the `Form1` class:

```
public Form1()
{
    InitializeComponent();

    this.panel1.AutoScrollMinSize = this.panel1.BackgroundImage.Size;
}
```

Now run the application again. Your form should now have scroll bars, as shown in [Figure 13-2](#). You will find that you can scroll around to view the entire image.

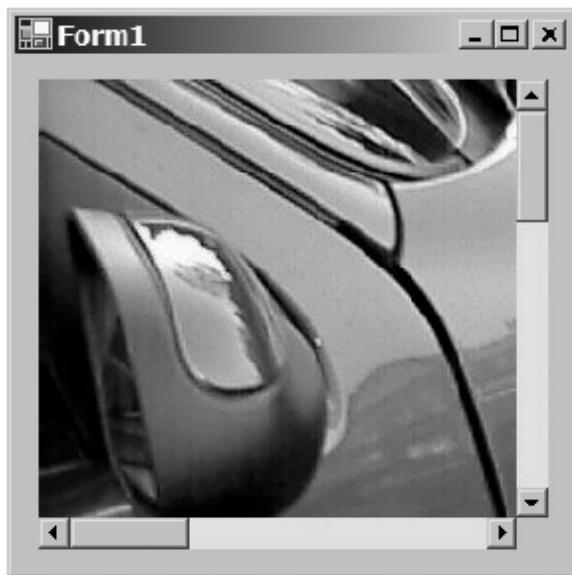


Figure 13-2: Scrolling picture in a Panel control

Setting the Scroll Bar Position

The scroll bars we added allow the user to scroll, but what if you want to control the positioning of the scroll bars? For example, if you are developing a word processor application, and you want to allow the user to search for some text, then presumably the program will respond to the search by moving the scroll bars so that the text in question is in the viewport.

We will develop the `ScrollingPicture` sample to demonstrate how to do this. Instead of having the scroll bars set to show the top-left area of the picture at startup, we will have the application start up showing the central part of the picture.

You can set (and retrieve) the position of the scroll bars using the `AutoScrollPosition` property. This property returns the coordinates of the top-left point of the viewport, relative to the top-left corner of the virtual space. For the example, this will be (0, 0).

It is not possible to set the scroll bar position until the form actually loads, so we must add an event handler for the `Load` event. Here is what it looks like:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    int midX = this.panel1.AutoScrollMinSize.Width/2;
    int midY = this.panel1.AutoScrollMinSize.Height/2;
    int halfSizeX = this.panel1.Size.Width/2;
    int halfSizeY = this.panel1.Size.Height/2;
    int startPosX = midX-halfSizeX;
    if (startPosX < 0) startPosX = 0;
    int startPosY = midY-halfSizeY;
    if (startPosY < 0) startPosY = 0;

    this.panel1.AutoScrollPosition = new Point(startPosX, startPosY);
}
```

The bulk of this code is devoted to working out what the scroll position should be to show the center of the picture. The formula takes the width of the virtual space minus the width of the control, divided by 2, for the horizontal scroll bar, and makes similar calculations for the vertical scroll bar. However, if this formula gives a value less than zero, we set the position to zero (a value less than zero indicates that the control size is greater than the size of the virtual space, so no scroll bars will be displayed anyway).

After adding this code, the form looks like [Figure 13-3](#) when it starts.



Figure 13-3: Scrolling picture with adjusted scroll bars

Previous

Next

Previous

Next

Drawing in a Viewport

The `ScrollingPicture` sample was particularly simple because the `Panel` class handled all the drawing itself. However, if you are doing the drawing in your code, you will need to take account of the position of the scroll bars in your drawing code.

For example, suppose you need to draw something—a string, a rectangle, or even an image using `Graphics.DrawImage()`—at the point (100, 100) in the virtual space. If the scroll bars are set so the viewport shows the top-left corner of the virtual space, then it is fine to draw the item at point (100, 100). But suppose the user has scrolled 50 pixels down and 30 pixels across, so that the `AutoScrollPosition` property of the control is (30, 50). Then the point (100, 100) in the virtual space corresponds to the point (70, 50) in the actual viewport, so the item actually needs to be drawn at location (70, 50). Although this sounds complicated, as you will see soon, this can be handled very simply with the `Graphics.TranslateTransform()` method.

In this section, we will develop another short example, called `ScrollingText`. This example is similar to the previous one, except that it displays the string "Hello, World" in four locations, which means we will be handling the drawing code ourselves. Also, the control will handle its own scrollability, rather than having it set by the containing form. [Figure 13-4](#) shows what the sample looks like.



Figure 13-4: Scrolling text

Start by creating a new Windows Control Library project called `ScrollingText`. (Since this is a completely new, separate example from the previous one, you can create a new solution and put the project in the new solution.) Next, rename the file to `ScrollingText.cs`, which also sets the `(Name)` property to `ScrollingText`.

Modify the code so that the control class is derived from `Panel`:

```
public partial class ScrollingText : System.Windows.Forms.Panel
```

Using the Properties window, set the `BackColor` property of the control to `System.Window`, the `AutoScroll` property to `True`, and the `AutoScrollMinSize` property to `(600, 400)`.

Now add a member field that will store the font for the text:

```
Font textFont = new Font("Times New Roman", 24);
```

Next, add the following event handler for the `Paint` event:

```
private void ScrollingText_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.TranslateTransform(
        this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
    g.DrawString("Hello, World", textFont, Brushes.Black, 40, 40);
    g.DrawString("Hello, World", textFont, Brushes.Red, 40, 240);
    g.DrawString("Hello, World", textFont, Brushes.Blue, 350, 40);
    g.DrawString("Hello, World", textFont, Brushes.Green, 350, 240);
}
```

The following is the key line of code here:

```
g.TranslateTransform(
    this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

This shifts the origin, as far as drawing operations are concerned, by the same amount as the control is currently scrolled by. This means that the text will appear in the correct location whatever the position of the scroll bars.

To test the control, you simply need to create a Windows Form application and add the control to the form.

Previous

Next

Previous

Next

The examples you have seen so far should give you an idea of how to have scrolling managed entirely by the Windows Forms classes supplied by Microsoft. You have also seen how easy it is to implement scrolling by simply setting a couple of properties on the control. So, you may be wondering why you would ever *not* want to code scrolling in that way. Actually, there are several reasons for choosing another technique.

Why use Alternative Scrolling Techniques?

Here are the main reasons to consider using alternative scrolling techniques:

To add scrolling support: Not all the controls in the `System.Windows.Forms` namespace support scrolling. In order to support scrolling, the control must be derived directly or indirectly from the class `System.Windows.Forms.ScrollableControl`. This is the case for many classes, including `TextBox`, `ListBox`, and the `Form` class itself, but there are exceptions. Most important, the class `Control` does not support scrolling. The `Panel` class does, which is why, in the previous example, we derived our custom control from `Panel` rather than `Control`. Of course, if you wish, there is nothing to stop you from deriving a custom control directly from `ScrollableControl`. The `Panel` class adds functionality to `ScrollableControl` that facilitates having constituent controls.

To implement smooth scrolling: You may want to implement *smooth scrolling*, which is a technique that is useful for scrolling by large amounts. When you click to move the scroll bar a big distance, instead of the viewport jumping instantaneously to the new position, as happens with the scrolling supplied by the Microsoft Windows Forms classes, smooth scrolling takes you there over an interval of perhaps half a second, giving your application a much more polished and professional appearance. For example, you might use this technique in a multiple-column list custom control. When the user scrolls up or down, instead of just having the list jump to the new location, you can incrementally scroll it by a few pixels at a time, and do the incremental scrolling quickly, so that it has the appearance of sliding to the new position. This provides the user with an additional visual clue about the navigation action just taken.

To gain low-level control: If you rely on a control deriving from `ScrollableControl`, and rely on its in-built scrolling capabilities, you don't get any low-level control over the scrolling or any actions that your control takes in response to a user scrolling. If you implement scrolling yourself, then you can access, for example, the events that occur when the user scrolls in order to provide your own handlers.

To handle unlimited virtual space: One scrolling technique involves having an in-memory drawing surface that represents the entire virtual space and transferring the appropriate portion of that image to the screen as required. Another technique involves simply drawing the contents of the viewport to the screen. Consider the first example in this chapter, which used a scrollable image. Even though we were viewing only a portion of the image at one time, the entire image had to remain in memory as the background image associated with the `Panel` control. On the other hand, in the second example, which involved drawing text in the `Paint` event, there was no such area of memory. The only record of what had been drawn was the screen itself. If the virtual space is limited in size, as it was in both of these examples, there is no problem with holding a representation of the entire virtual space in memory. However, for many applications, such as spreadsheets or word processors, the size of virtual space is potentially unlimited. In this case, it is not feasible to use a drawing technique in which the virtual space is permanently represented in memory.

If you do not want to use the `ScrollableControl` scrolling support, your usual technique will involve adding horizontal and vertical scroll bars to your control from the Toolbox. These are represented by the `HScrollBar` and `VScrollBar` classes. Then you can add handlers to the various scrolling events raised by these classes in order to implement scrolling.

Which Approach should you Use?

The approach that you take to building a scrolling control depends on the capabilities that you want:

Do you need to compose this custom control of other Windows Forms controls? If you are composing this custom control of other controls (such as buttons, text boxes, and so on), you don't need to draw into your custom control using GDI+, and the virtual space is limited in size, you should build your custom control by deriving from the `UserControl` class, which derives from `ScrollableControl`. If you add child controls to your custom control, these are called *constituent controls*, as described in [Chapter 12](#).

Do you need to draw using GDI+ into the custom control? If you need to draw into your custom control using GDI+, you are not composing this custom control with other controls, the virtual space is limited in size, and you don't want to implement smooth scrolling, then you don't even need to build a custom control. You can create an instance of a PictureBox control, place it in a Panel control, draw into an image (as demonstrated in [Chapter 5](#)), and set the `BackgroundImage` property of your custom control to your just-built image. Alternatively, you can, as in earlier examples, draw directly into a Panel or some similar control. If you need to both draw into your virtual space and add constituent controls to it, and it is limited in size, you derive from the `UserControl` class, build an image, draw into the image, and set the `BackgroundImage` property of your custom control to your built image.

Do you want to create a smooth scrolling virtual space? If you develop an application that scrolls smoothly, it will have the appearance of sliding rather than jumping. If you want to implement smooth scrolling, you derive from the `UserControl` class and add horizontal and vertical scroll bars to your custom control. You build an image, draw into the image, and manage the drawing of the image in your custom control.

Is the virtual space very large, such as you might use for a mapping custom control? In a mapping custom control, users may be able to pan over a very large area, and it may not be practical to create an image that contains the entire area that could be panned over. For this, you derive from the `UserControl` class and add horizontal and vertical scroll bars. Instead of drawing to an image, you respond directly to Paint events and manage the position of the viewport yourself.

The techniques for drawing into a custom control and using smooth scrolling are demonstrated in the following sections.

 Previous

Next 

 Previous

Next 

Using a Panel to Scroll another Control

Let's consider how to handle scrolling with a control that does not support scrolling. In the case where you draw to a control that does not support scrolling, such a PictureBox, you can embed this control in a control that does support scrolling. This method is quite simple, and you probably already know how to implement it.

To demonstrate this technique, we will work through an example that will look identical to the `ScrollText` example: it will display "Hello, World" in several locations in the virtual space and allow the user to scroll between them. Instead of drawing to a scrollable control, however, we will draw to a PictureBox and put that control in a Panel control. We will arrange that, when the form is loaded, it draws the text into a bitmap, and then loads this bitmap into the Picture-Box control.

Doing things this way can improve performance when scrolling, since it means that when the user scrolls, there is no need to redraw the new part of the viewport that appears; it has already been drawn, so the image simply needs to be transferred from the bitmap in memory. Transferring images from one bitmap to another is known as a *BitBlt* (Bit Block Transfer) operation, which we will explore in the [next chapter](#). However, you do need to know that this technique stores the image for the entire virtual space, so it should be used only when the virtual space is limited in size.

To start this example, add a new Windows Application to the solution and name it `PictureBoxScrollText`. Drag a Panel control onto the form. Using the Properties window, set the `AutoScroll` property of the Panel to True and `AutoScrollMinSize` to `(600, 400)`.

Now drop a PictureBox control into the Panel control (in the actual Panel, not in the main form) and set its `Dock` property to `Fill`, so that it occupies the entire area of the Panel. Use the Properties window to set the `BackColor` of the PictureBox to `SystemWindow`.

Now we can modify the code for the `Form1` class. First, add the following field:

```
public class Form1 : System.Windows.Forms.Form
{
    Font textFont = new Font("Times New Roman", 24);
```

Add a Load event handler for the `Form` and modify it as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Bitmap b = new Bitmap(600, 600);
    Graphics g = Graphics.FromImage(b);
    g.FillRectangle(
        Brushes.White, new Rectangle(0, 0, b.Width, b.Height));
    g.DrawString("Hello, World", textFont, Brushes.Black, 40, 40);
    g.DrawString("Hello, World", textFont, Brushes.Red, 40, 240);
    g.DrawString("Hello, World", textFont, Brushes.Blue, 350, 40);
    g.DrawString("Hello, World", textFont, Brushes.Green, 350, 240);
    pictureBox1.BackgroundImage = b;
    pictureBox1.Size = b.Size;
}
```

This drawing code is very similar to that for the earlier `ScrollText` example, except that we no longer need to set the `TranslateTransform()` method for the `Graphics` object.

Build and run the application. You'll see that you can pan over the image that you drew. If the state of the application changes, you can draw again into the same image, or create a new image, draw into it, and set the new image on the `PictureBox` control.

 Previous

Next 

 Previous

Next 

Smooth Scrolling

Creating a smooth scrolling virtual space can give your application a very polished appearance. In addition to better aesthetics, smooth scrolling adds to the functionality of an application in that it gives the users one more visual clue about the navigation action just taken. They press the down arrow key, the content of the custom control slides up, and this subtly confirms that they pressed the key that they wanted to press. If the display simply jumps to a new position, the visual clue indicates only that they have moved; other than the new position of the scroll bar, there is no clue to confirm that the movement was up as opposed to down.

In this section, we will work through an example that illustrates smooth scrolling. We will implement smooth scrolling by creating an image that contains the graphical contents of our custom control and sliding the image appropriately behind a viewport. The approach involves the following main tasks:

- Create a custom control that derives from `UserControl`, and add horizontal and vertical scroll bars as constituent controls.
- Write a `Paint` event handler that draws the image as appropriate in the custom control. In addition, the `Paint` event handler draws the background color into any areas of the custom control not covered by the image.

When sliding the image, we need to draw the image numerous times at appropriate positions and times. We don't want to implement this using a process of invalidation and waiting for the `Paint` event to come through for each drawing of the image. Rather, we want to have a loop that does the drawing, and then uses the `Sleep()` method of the `Thread` class for each iteration of the loop. Once we involved the event queue, as we would if we were to invalidate and wait for the `Paint` event, the system performance would be indeterminate. There would be no guarantee that the `Paint` event would come through on schedule, and this could cause the slide to appear less smooth.

Creating the Smooth Scrollable Control

The smooth scrollable control looks similar to the previous examples. It displays the text "Hello, World" in several locations in the viewport, as shown in [Figure 13-5](#).



Figure 13-5: A smooth scrollable control

To code the example, create a new Windows Control Library and call it `SmoothScrollableControl`. Rename `UserControl1.cs` to `SmoothScrollableControl.cs`.

Note that because we are implementing the scrolling ourselves, we are not using the scrolling capabilities of `UserControl`. The application would work the same way whether we derive from `UserControl`, `Panel`, or `Control`. However, deriving from `UserControl` gives us a slightly more user-friendly interface in the Design view in Visual Studio .NET. Therefore, we will keep our control as derived from `UserControl`.

Now use the Toolbox to add `HScrollBar` and `VScrollBar` controls to `SmoothScrollableControl`, and set the Dock property to `Bottom` and `Right`, respectively. Use the Properties window to rename these scroll bars to `hScrollBar` and `vScrollBar`, respectively.

Next, we need to make some changes to the code for the control. First, add a `System.Threading` namespace reference. We will need the `System.Threading.Thread` class to control the timing for the smooth scrolling:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using System.Threading;
```

Now add the following fields to the class:

```
public class SmoothScrollableControl : System.Windows.Forms.UserControl
{
    private Image scrollingImage;
    private Point originOfImage = new Point(0, 0);
```

`scrollingImage` will contain the bitmap for the virtual space, and `originOfImage` stores the coordinates of the viewport.

We will add a public property to allow client code to access the scrolling image:

```
public Image ScrollingImage
{
    get
    {
        return scrollingImage;
```

```
    }
    set
    {
        scrollingImage = value;
        adjustScrollBars();
    }
}
```

Also, add another property that gives us the area of the control that is actually available for drawing our image in; that is, the client area minus the area taken up by the scroll bars:

Also modify the constructor as follows:

```
public SmoothScrollableControl()
{
    InitializeComponent();

    this.SetStyle(ControlStyles.Opaque, true);
}
```

Setting the style to `ControlStyles_Opaque` makes drawing more efficient and prevents flickering by preventing the control from drawing a background color. There is no need for a background color, since our bitmap and the scroll bars will together cover the viewport entirely.

Next, we will look at the process of painting. First of all, here is a helper method that works out the portion of the client area corresponding to the viewport (as we have just seen, that is the client area excluding the scroll bars), and draws the appropriate portion of the bitmap to just this area:

```
private void drawImage(Graphics g, int x, int y)
{
    g.SetClip(ScrollingImageRectangle);

    Rectangle drawRect =
        new Rectangle(new Point(x, y), ScrollingImageRectangle.Size);
    g.DrawImage(this.scrollingImage,
        ScrollingImageRectangle, drawRect, GraphicsUnit.Pixel);
}
```

Writing the Paint Event Handler

Now let's examine the `Paint` event handler. Note that this handler is not called when the user scrolls. As you will soon see, the event handlers for scrolling events call the `drawImage()` helper method directly to update the display. This event handler is called whenever there is some other reason why the control needs to be repainted (for example, it has just been restored from a minimized state). It deals with copying the relevant portion of the bitmap to the screen, but also whites out any other areas to the right of and below the image that might need repainting.

```
private void SmoothScrolling_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Brush bgb = new SolidBrush(this.BackColor);

    if (this.scrollingImage != null)
    {
        drawImage(g, this.originOfImage.X, this.originOfImage.Y);

        g.ResetClip();
    }
}
```

```

Rectangle r;

// Paint area to right of image and left of the vertical scroll bar
r = new Rectangle(scrollingImage.Width, 0,
    this.vScrollBar.Left scrollingImage.Width,
    this.ScrollingImageRectangle.Height);
if (! r.IsEmpty)
    g.FillRectangle(bgb, r);

// Paint area below image and above horizontal scroll bar
r = new Rectangle(0, scrollingImage.Height,
    this.ScrollingImageRectangle.Width,
    this.hScrollBar.Top - scrollingImage.Height);
if (! r.IsEmpty)
    g.FillRectangle(bgb, r);

// Paint the small rectangle at the bottom right of the control
r = new Rectangle(this.hScrollBar.Right, this.vScrollBar.Bottom,
    this.ClientRectangle.Width - this.hScrollBar.Right,
    this.ClientRectangle.Height - this.vScrollBar.Bottom);
g.FillRectangle(bgb, r);

}

else
{
    g.FillRectangle(bgb, this.ClientRectangle);
}
}
}

```

Let's look at the code that handles the scroll bars themselves.

Method for Adjusting the Scroll Bars

First up is a helper method that calculates the correct lengths for the bars and sets various properties on them that describe how far they should be allowed to scroll:

```

private void adjustScrollBars()
{
    if (scrollingImage != null)
    {
        hScrollBar.Minimum = 0;
        hScrollBar.Maximum = scrollingImage.Width;
        hScrollBar.LargeChange = ScrollingImageRectangle.Width;
        hScrollBar.SmallChange = ScrollingImageRectangle.Width / 10;
        hScrollBar.Value = originOfImage.X;
        vScrollBar.Minimum = 0;
        vScrollBar.Maximum = scrollingImage.Height;
        vScrollBar.LargeChange = ScrollingImageRectangle.Height;
        vScrollBar.SmallChange = ScrollingImageRectangle.Height / 10;
        vScrollBar.Value = originOfImage.Y;
    }
}

```

Method for Sliding the Scroll Bars

The scrolling event handlers, which we will look at soon, both call another helper method, `slide()`. This helper method handles the process of sliding the scroll bars from a given old position to a given new position (the positions being stored as the coordinate of the top-left corner of the viewport, relative to the top-left corner of the virtual space). The `slide()` method is the one where the smooth scrolling action actually takes place. It enters a loop. At each iteration of the loop, we redraw the viewport in an incrementally scrolled position, then use the `Thread.Sleep()` method to wait for 10ms (0.01 of a second) before moving to the next iteration.

```

private void slide (Point oldP, Point newP)
{

```

```
Graphics g = this.CreateGraphics();

if (oldP.X != newP.X)
{
    // Slide horizontally
    bool goingUp = newP.X - oldP.X > 0;

    int delta = (newP.X - oldP.X) / 12;
    if (goingUp && delta < 1)
        delta = 1;
    if (!goingUp && delta > -1)
        delta = -1;
    int i = oldP.X;
    while (true)
    {
        if (i == newP.X)
            break;
        i += delta;
        if (goingUp && i > newP.X)
        {
            i = newP.X;
            continue;
        }
        if (!goingUp && i < newP.X)
        {
            i = newP.X;
            continue;
        }
        drawImage(g, i, oldP.Y);
        if (i != newP.X)
            Thread.Sleep(10);
    }
}
if (oldP.Y != newP.Y)
{
    // Slide horizontally
    bool goingUp = newP.Y - oldP.Y > 0;
    int delta = (newP.Y - oldP.Y) / 12;
    if (goingUp && delta < 1)
        delta = 1;
    if (!goingUp && delta > -1)
        delta = -1;
    int i = oldP.Y;
    while (true)
    {
        if (i == newP.Y)
            break;
        i += delta;
        if (goingUp && i > newP.Y)
        {
            i = newP.Y;
            continue;
        }
        if (!goingUp && i < newP.Y)
        {
            i = newP.Y;
            continue;
        }
        drawImage(g, oldP.X, i);
        if (i != newP.Y)
            Thread.Sleep(10);
    }
}
```

Handlers for Scrolling Events

Now we move on to the event handlers for the scrolling events. To add the vertical scrolling event handler, click the vertical scroll bar in the Design window in Visual Studio .NET. Then use the Properties window to locate the `Scroll` event and double-click it to add an event handler.

The `Scroll` event is raised whenever the user clicks the scroll bars to scroll. Its handler is passed a `ScrollEventArgs` argument. `ScrollEventArgs` supplies a property, `Type`, which indicates the nature of the scroll event. `ScrollEventType.SmallIncrement` and `ScrollEventType.SmallDecrement` indicate that the user clicked one of the small arrows at the ends of the scroll bars to perform an incremental scroll. The `LargeIncrement` and `LargeDecrement` values indicate that the user clicked in the light area in the scroll bar to move the scroll bar by a large amount, or actually clicked and dragged the scroll bar itself.

Add the following code for the `Scroll` event handler for the vertical scroll bar:

```
private void vScrollBar_Scroll(
    object sender, System.Windows.Forms.ScrollEventArgs e)
{
    switch (e.Type)
    {
        case ScrollEventType.SmallIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.SmallDecrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeDecrement:
            Point oldOrigin = originOfImage;
            originOfImage = new Point(originOfImage.X, e.NewValue);
            slide(oldOrigin, originOfImage);
            break;
        default:
            originOfImage = new Point(originOfImage.X, e.NewValue);
            this.Invalidate();
            break;
    }
}
```

Similarly, add this code for the horizontal scroll bar:

```
private void hScrollBar_Scroll(
    object sender, System.Windows.Forms.ScrollEventArgs e)
{
    switch (e.Type)
    {
        case ScrollEventType.SmallIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.SmallDecrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeDecrement:
            Point oldOrigin = originOfImage;
            originOfImage = new Point(e.NewValue, originOfImage.Y);
            slide(oldOrigin, originOfImage);
            break;
        default:
            originOfImage = new Point(e.NewValue, originOfImage.Y);
            this.Invalidate();
            break;
    }
}
```

Resize Event Handler

We have one final method to add, which is an event handler for the `Resize` event on the `SmoothScrollableControl` control. If the control is resized, we will need to recalculate the scroll bar positions and possibly redraw the viewport.

```
private void SmoothScrollableControl_Resize(  
    object sender, System.EventArgs e)  
{  
    if (scrollingImage != null)  
    {  
        int x = Math.Min(originOfImage.X,  
            scrollingImage.Width - ScrollingImageRectangle.Width);  
        x = Math.Max(x, 0);  
        int y = Math.Min(originOfImage.Y,  
            scrollingImage.Height - ScrollingImageRectangle.Height);  
        y = Math.Max(y, 0);  
        originOfImage = new Point(x, y);  
    }  
    adjustScrollBars();  
    this.Invalidate();  
}
```

You can now build the control and add it to the Toolbox.

Testing the Control

Add a new Windows Application called `TestSmoothScrollableControl` to the project and make it the startup project. Add a reference to the custom control in the test application and place a `SmoothScrollableControl` on the form's surface.

The `SmoothScrollableControl` class itself implements all the functionality required to display a virtual space in a viewport with smooth scrolling. Beyond that though, it is a general-purpose class. It doesn't specify the image to go in the virtual space. It is up to the containing form to do that. So, we need to add a `Load` event to the form and modify the code as follows:

```
private void Form1_Load(object sender, System.EventArgs e)  
{  
    Bitmap b = new Bitmap(600, 600);  
    Graphics g = Graphics.FromImage(b);  
    g.FillRectangle(Brushes.White,  
        new Rectangle(0, 0, b.Width, b.Height));  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Black, 40, 40);  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Red, 350, 40);  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Blue, 40, 240);  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Green, 350, 240);  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Aquamarine, 40, 440);  
    g.DrawString("Hello, World",  
        new Font("Times New Roman", 24), Brushes.Magenta, 350, 440);  
    smoothScrollableControl1.ScrollingImage = b;  
}
```

Now we can build and run the application. Then, click the scroll bars to see the smooth scrolling. Pressing the arrow keys on the keyboard also causes smooth scrolling to take place.

 Previous

Next 

 Previous

Next 

Summary

In this chapter, you saw several methods for creating a scrolling virtual space. In almost all of these cases, it is far easier to create a scrolling virtual space if you use custom controls to implement it.

You saw how to build virtual spaces with the following characteristics:

- The virtual space can contain a number of constituent controls. You implement this by creating a custom control and deriving from the `UserControl` class.
- If you want also to draw graphics into a virtual space that contains constituent controls, you can draw into an image, and set the `BackgroundImage` property for your custom control.
- If you have only graphics, and the size of your virtual space is limited, you don't even need to make a custom control. You can draw into an image, set the `BackgroundImage` property for a `PictureBox` control, put the `PictureBox` control into a `Panel` control, and set some properties on the `Panel` control.
- If you want to create a smooth scrolling virtual space, you derive from the `UserControl` class, and handle the scroll bar events yourself. You worked through a sample implementation in this chapter.

There is one more topic that is vital to building custom controls. It's important that the user be able to operate your custom control with the mouse. In the [next chapter](#), you will see how to handle mouse events and cursors.

 Previous

Next 

 PreviousNext 

Chapter 14: Mouse Events and Cursors

Overview

Most custom controls allow the user to interact with custom controls using the mouse. It would be hard to imagine a custom control that didn't allow the user to manipulate it with the mouse. Programming to enable mouse interaction is easy, but there are a few important details.

Most programmers reading this book are familiar with mouse events and probably have written event handlers for them. Therefore, we won't spend much time on the simplest of mouse event handling, but will instead concentrate on the more advanced issues, primarily those that pertain to building custom controls. For the most part, handling mouse events in Windows Forms is similar to handling mouse events in MFC and other toolkits. There are not a lot of surprises, but there are a few.

This chapter begins with a brief recap of the mouse events available in .NET. Then it covers mouse cursors, hit testing, and mouse-event routing. We'll also look at drawing while dragging and drag-scrolling, after an introduction to BitBlt.

 PreviousNext  PreviousNext 

Mouse Events

There are seven mouse events, as listed in [Table 14-1](#). These are events defined in the `Control` class. Most Windows Forms classes, including the `Form` class, derive from the `Control` class.

Table 14-1: Lower-Level Mouse Events

Event	Description
MouseDown	Raised when the cursor is over a control and the user presses a mouse button.
MouseEnter	Raised when the cursor enters a control while no mouse button is pressed.
MouseHover	Raised when the mouse cursor hovers over a control.
MouseLeave	Raised when the mouse cursor leaves a control.
MouseMove	Raised when the mouse cursor moves over a control.
MouseUp	Raised when the mouse button is released. By default, this event always goes to the control in which the mouse button was pressed.
MouseWheel	Raised when the user moves the mouse wheel while the control has the focus.

If the user moves the mouse cursor into a control and takes some action, the normal sequence of events, depending on exactly what the user does, runs something like `MouseEnter`, `MouseMove`, `MouseHover`/`MouseDown`/`MouseWheel`, `MouseUp`, `MouseLeave`.

In addition, it's worth being aware of the higher-level mouse events listed in [Table 14-2](#).

Table 14-2: Higher-Level Mouse Events

Event	Description
Click	User has clicked the mouse button.
DoubleClick	User has double-clicked the mouse button.

These higher-level events each encapsulate several lower-level events. For example, the `Click` event is raised when the user has depressed and then released the left mouse button. These events can be

useful for simple work. However, if you want your application to behave in a sophisticated manner with regard to mouse events, you will probably want to use the lower-level events such as `MouseDown` to gain a finer degree of control. Accordingly, we will focus on the lower-level events in this chapter.

Note In this chapter, I will routinely refer to the *mouse button* or the *left mouse button*. In both cases, this means the mouse button that is used for click events. With the default Windows settings, this will be the left mouse button though, obviously, if you have set up your computer for a left-handed person, this will be the right button. We will not consider context menu events or the context mouse button (normally the right button) at all in this chapter.

Previous

Next

Previous

Next

Mouse Cursors

One of the key design principles of graphical user interfaces is that you always want to give the user as much visual feedback as possible. Mouse cursors are a great way to provide visual clues for them. There are a variety of stock mouse cursors that you can use to help your application's users. In addition, you can create custom mouse cursors to fit the style and use of your custom control.

Every mouse cursor has what is called the *hot spot*. This is the exact point in the cursor bitmap that corresponds to the mouse's actual location; that is, the location that will be passed to event handlers as the mouse location. For the standard arrow cursor, the hot spot is the tip of the arrow. The hot spot in an I-beam cursor is a point on the I-beam, just above the bottom end, as illustrated in [Figure 14-1](#).

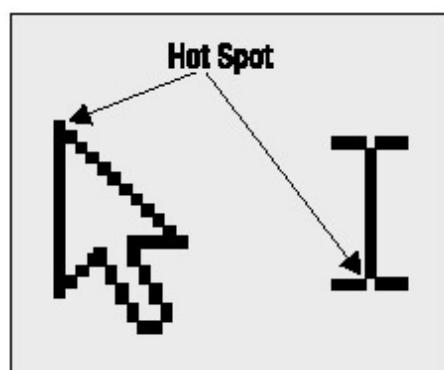


Figure 14-1: Hot spots on arrow and I-beam cursors

It is important to be aware that the hot spot is not necessarily at the top left of the cursor. It could theoretically be any point within the area of the cursor, and it will be in different positions for different cursors. As shown in [Figure 14-1](#), the standard arrow cursor gives a very clear visual clue to the user about where the hot spot is, but that isn't the case for the I-beam cursor.

The `Cursor` class encapsulates mouse cursor functionality. You can create a cursor from a file or a Stream. In addition, there are a number of standard cursors that you can get from static properties of the `Cursors` class. If you set the cursor to `Cursors.Default`, the cursor is the default cursor, which is usually the arrow cursor.

Using Standard Mouse Cursors

A large number of standard mouse cursors are available. You can obtain any of these via static properties of the `Cursors` class. For example, to set the cursor of the current control (which we will assume is accessed via the `this` reference) to a hand, you can use the following:

```
this.Cursor = Cursors.Hand;
```

To set the cursor to a cross, you would code the following:

```
this.Cursor = Cursors.Cross;
```

The following are the `Cursors` properties that return different cursors:

AppStarting	Arrow	Cross	Hand
Help	HSplit	IBeam	No
NoMove2D	NoMoveHoriz	NoMoveVert	PanEast
PanNE	PanNorth	PanNW	PanSE
PanSouth	PanSW	PanW	SizeAll
SizeNESW	SizeNS	SizeNWSE	SizeWE
UpArrow	VSplit		WaitCursor

In addition, a property called `Cursors.Default` returns the default cursor.

Changing Mouse Cursors

The following example demonstrates the use of standard cursors. It is a simple Windows application that displays four rectangles. Each rectangle is associated with a different cursor, and as the mouse is moved over each rectangle, the cursor changes accordingly. If you click the mouse, the wait cursor is displayed for one second, and processing of mouse events is suspended for that time. [Figure 14-2](#) shows the application in action.

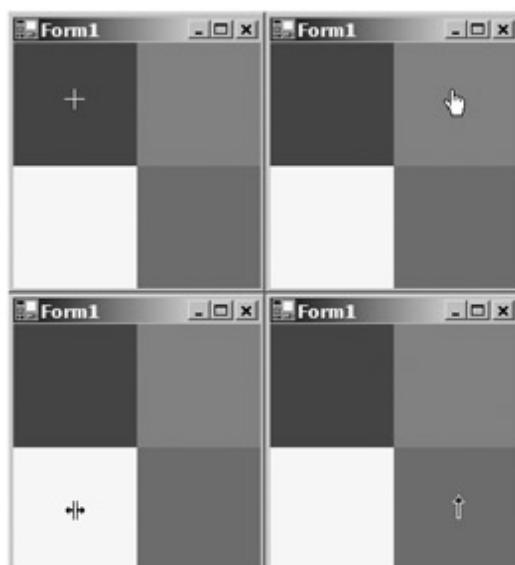


Figure 14-2: Changing mouse cursors

To start this example, create a new Windows Application called `CursorsExample`. Add a `MouseMove` event handler to the form, and modify it like this:

```
private void Form1_MouseMove (object sender, System.EventArgs e)
{
    Point p = new Point(e.X, e.Y);
    Rectangle r;

    if (new Rectangle(0, 0, 100, 100).Contains(p))
        this.Cursor = Cursors.Cross;
    else if (new Rectangle(100, 0, 100, 100).Contains(p))
        this.Cursor = Cursors.Hand;
    else if (new Rectangle(0, 100, 100, 100).Contains(p))
        this.Cursor = Cursors.VSplit;
    else if (new Rectangle(100, 100, 100, 100).Contains(p))
        this.Cursor = Cursors.UpArrow;

    else
        this.Cursor = Cursors.Default;
}
```

Note that because the `MouseMove` event is raised so often, it is very important to do as little processing as possible in it. In this case, the handler looks a bit dubious at first sight because we are continually setting the mouse cursor. However, this is not a problem, since all we are doing is setting one reference.

Now add a `Paint` event handler and modify it like this:

```
private void Form1_Paint(  
    object sender, System.Windows.Forms.PaintEventArgs e)  
{  
    Graphics g = e.Graphics;  
    g.FillRectangle(Brushes.Blue, 0, 0, 100, 100);  
    g.FillRectangle(Brushes.Red, 100, 0, 100, 100);  
    g.FillRectangle(Brushes.Yellow, 0, 100, 100, 100);  
    g.FillRectangle(Brushes.Green, 100, 100, 100, 100);  
}
```

Run the application. You'll see that as you move the mouse over the window, the mouse cursor changes. If you move the mouse over the area to the right of the colored squares, the mouse cursor changes to the default cursor. If you move the mouse over the window borders, the cursor changes to the appropriate resize cursor.

Changing to the `WaitCursor` for a Specified Period

Sometimes, you may want to change the cursor to the `WaitCursor` for the entire application for a certain amount of time, perhaps when doing an extensive amount of processing. To do this, you can set the static `Cursor.Current` property to `Cursors.WaitCursor`. Setting `Cursor.Current` is different from setting the `Cursor` property of the current control for two reasons:

- This assignment in `Cursor.Current` is effective only until the current event handler returns, or until your code calls the `Application.DoEvents()` method.
- Assigning `Cursor.Current` to any value other than the default cursor, `Cursor.Default`, prevents processing of mouse events while the temporary cursor is in place.

This behavior is what you want: You can set the cursor to the `Cursors.WaitCursor`, and then forget about it. After the processing is done, you return from your event handler, and the cursor reverts to the correct cursor. To see this functionality, add the following two lines to the `Click` event handler for the form:

```
private void Form1_Click(object sender, System.EventArgs e)  
{  
    Cursor.Current = Cursors.WaitCursor;  
    System.Threading.Thread.Sleep(1000);  
}
```

This changes the cursor to the `WaitCursor` for one second.

Creating Custom Mouse Cursors

When you're building a custom control, you might find that none of the stock cursors meets your needs. For example, you might have a mapping custom control where you allow the user to zoom in and out. When the control is in the zoom in/out mode, you want to change the mouse cursor to give the users a visual indication that something special will happen when they click. If you need a special kind of cursor, you can create a custom one.

The easiest way to build a custom cursor is to start with an existing cursor. You can get an existing cursor by searching for files with a `.cur` extension in the Microsoft Visual Studio .NET directory on the drive where your Visual Studio .NET IDE is installed.

As an example of a custom cursor, the cursor named `MyCursor.cur`, shown in [Figure 14-3](#), has certain pixels that are defined to be transparent. All pixels other than the black-and-white pixels are transparent. The hot spot is the point in the middle. When viewed in the image editor in Visual Studio .NET, the transparent pixels are blue-green.

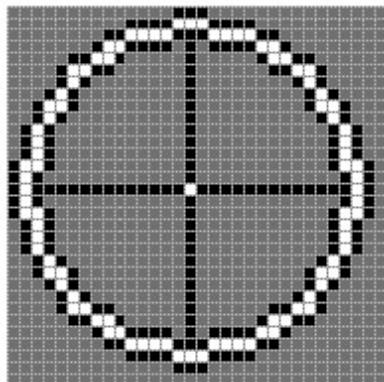


Figure 14-3: A custom cursor

To see the cursor in action, place the `MyCursor.cur` file (available with the rest of the downloadable code for this book, at the Apress web site: www.apress.com) in the directory where your application will execute (in other words . . .\CursorExample\bin\Debug). Then modify the previous example of using standard cursors as follows:

```
private void Form1_MouseMove(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    Point p = new Point(e.X, e.Y);
    Rectangle r;
    if (new Rectangle(0, 0, 100, 100).Contains(p))
        this.Cursor = new Cursor("MyCursor.cur");
    else if (new Rectangle(100, 0, 100, 100).Contains(p))
        this.Cursor = Cursors.Hand;
    .
    .
    .
}
```

Run the example and move the mouse over the upper-left rectangle to see the custom cursor.

You can also modify a cursor's hot spot using the Set Hot Spot Tool, which is a button on the toolbar that is visible when you are in the image editor:



◀ Previous

Next ▶

◀ Previous

Next ▶

Hit Testing and Drawing During Mouse Events

When users are moving the mouse over your custom control (without the mouse button pressed), you can alter the appearance of your custom control or change the mouse cursor to give users a clue that they can take some action at certain locations in your custom control. When the user presses or releases the mouse button, you use *hit testing* to determine what action you should take.

Hit testing is the process where you determine if a particular point is in a given rectangle, region, or path. As the user moves the mouse around the custom control, you want to know when the mouse is within certain regions.

You saw an example of hit testing in the `CursorsExample` example earlier in this chapter, where we used the `Rectangle.Contains()` method:

```
if (new Rectangle(0, 0, 100, 100).Contains(p))
    this.Cursor = Cursors.Cross;
```

When you are hit testing in regions or paths, as opposed to rectangles, you use the `Region.Visible()` or `GraphicsPath.Visible()` method, which tests whether a point is located inside the given region or graphics path, respectively.

OPTIMIZING GRAPHICS CODE

With today's fast computers, there is a rule of thumb that you should not prematurely optimize your code. A lot of unnecessary ugliness has been introduced into otherwise clean code in the name of optimization. This tendency is understandable given that many of us came from slow computing environments in the past where, if we didn't proactively optimize, the application would be unusable. Now, primarily we strive to create the cleanest and most maintainable code possible. We do any necessary optimization *after* we have implemented the main functionality. However, I make an exception when writing graphical code, where I attempt to draw exactly what is necessary and no more. I have two main reasons for this philosophy:

- Once we involve the graphics subsystem, performance is not deterministic. An operation that runs quickly on one computer with a sophisticated graphics card runs slowly on another. The memory available can be important here, too. This is compounded by object-oriented techniques where we build abstractions on abstractions, and the farther away we get from the lowest-level abstractions, the more we forget the actual processing cost.
- When we involve the graphics subsystem, behavior is not deterministic. Where one graphics card may not mind at all if we fill a particular rectangle many times, another may cause flickering.

Therefore, I write graphics code with a bit of optimization. Primarily, this involves keeping state and drawing only if the state changes.

Hit Testing in Areas

The next example demonstrates hit testing on the three primary data types that describe areas. In addition, it demonstrates keeping state and drawing only if the state changes.

The hit testing example consists of a form in which three shapes are drawn. These shapes change color as the mouse moves, as shown in [Figure 14-4](#).

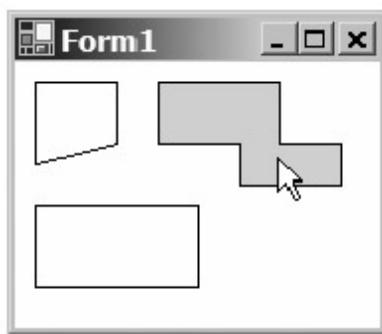


Figure 14-4: Hit testing

Whenever the mouse enters any shape, that shape is colored—pink for the double-rectangle shape, blue for the trapezoid, and green for the lower rectangle. As the mouse moves out of each shape, the shape's color is restored to white.

Create a new Windows Application and call it `HitTestExample`. Modify the `using` statements section to look like the following:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

```
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

Add the following private member variables to the `Form1` class:

```
private GraphicsPath path1;
private GraphicsPath path2;
private Region reg;
private Rectangle rect;
private bool inPath = false;
private bool inRegion = false;
private bool inRect = false;
```

`path1` represents the trapezoid, and `path2` represents the double rectangle. `rect` will store the coordinates of the lower rectangle. `reg` will represent the double rectangle. We use a `Region` object just to illustrate the use of the `Region` class, although we will need to construct it using the `GraphicsPath` `path2`. The `bool` variables simply indicate whether the mouse was in each of the shapes the last time the `MouseMove` event was raised.

Modify the constructor to initialize the member variables:

```
public Form1()
{
    InitializeComponent();
    path1 = new GraphicsPath();
    path1.AddLine(10, 10, 50, 10);
    path1.AddLine(50, 10, 50, 40);
    path1.AddLine(50, 40, 10, 50);
    path1.CloseFigure();
    path2 = new GraphicsPath();
    path2.AddLine(70, 10, 130, 10);
    path2.AddLine(130, 10, 130, 40);
    path2.AddLine(130, 40, 160, 40);
    path2.AddLine(160, 40, 160, 60);
    path2.AddLine(160, 60, 110, 60);
    path2.AddLine(110, 60, 110, 40);
    path2.AddLine(110, 40, 70, 40);
    path2.CloseFigure();
    reg = new Region(path2);
    rect = new Rectangle(10, 70, 80, 40);
}
```

This code simply creates the shapes. As just mentioned, you can construct a region only from a path, which is why we create and initialize the `path2` variable, then construct the `region` variable from it.

Next, add a `Paint` event handler, which will draw the shapes in white, like this:

```
private void Form1_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    g.DrawPath(Pens.Black, path1);
    g.DrawPath(Pens.Black, path2);
    g.DrawRectangle(Pens.Black, rect);
}
```

Finally, add a `MouseMove` event handler as follows. This is the handler where all the work is done.

```
private void Form1_MouseMove(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    Graphics g = this.CreateGraphics();

    try
    {
        if (inPath)
```

```
{  
    if (! path1.IsVisible(e.X, e.Y))  
    {  
  
        g.FillPath(Brushes.White, path1);  
        g.DrawPath(Pens.Black, path1);  
        inPath = false;  
        return;  
    }  
}  
}  
else if (inRegion)  
{  
    if (! reg.IsVisible(e.X, e.Y))  
    {  
        g.FillRegion(Brushes.White, reg);  
        g.DrawPath(Pens.Black, path2);  
        inRegion = false;  
        return;  
    }  
}  
}  
else if (inRect)  
{  
    if (! rect.Contains(e.X, e.Y))  
    {  
        g.FillRectangle(Brushes.White, rect);  
        g.DrawRectangle(Pens.Black, rect);  
        inRect = false;  
        return;  
    }  
}  
if (! inPath)  
{  
    if (path1.IsVisible(e.X, e.Y))  
    {  
        g.FillPath(Brushes.LightBlue, path1);  
        g.DrawPath(Pens.Black, path1);  
        inPath = true;  
        return;  
    }  
}  
}  
if (! inRegion)  
{  
    if (reg.IsVisible(e.X, e.Y))  
    {  
        g.FillRegion(Brushes.Pink, reg);  
        g.DrawPath(Pens.Black, path2);  
        inRegion = true;  
        return;  
    }  
}  
}  
if (! inRect)  
{  
  
    if (rect.Contains(e.X, e.Y))  
    {  
        g.FillRectangle(Brushes.LightGreen, rect);  
        g.DrawRectangle(Pens.Black, rect);  
        inRect = true;  
        return;  
    }  
}  
}  
}  
finally  
{  
    g.Dispose();  
}
```

```
}
```

The `MouseMove` event handler looks complex because it draws only what is necessary; it doesn't redraw with every `MouseMove` event. It draws only if the state changes.

Notice that all the code is contained in a `try...finally` block. This is to ensure that the `Graphics` object is disposed of correctly, as there are multiple `return` statements in the code. The presence of the `finally` block will ensure that whenever the computer hits a `return` statement in the `try` block, processing will transfer to the `finally` block before the handler exits.

We start off in the routine by examining the `inPath` field that tells us if the mouse was previously inside the trapezoid shape; in which case, that shape will be currently colored. If this is the case, we check whether the new mouse position is outside the trapezoid. If it is, we redraw the trapezoid white. Since the mouse can't jump between shapes, in this case, it must be outside the borders of the other shapes, so we know there is nothing that needs to be done with the other shapes and we can return:

```
if (! inPath)
{
    if (path1.IsVisible(e.X, e.Y))
    {
        g.FillPath(Brushes.LightBlue, path1);
        g.DrawPath(Pens.Black, path1);
        inPath = true;
        return;
    }
}
```

The rest of the code simply looks at all the remaining possibilities for where the mouse can be. Notice how we have carefully optimized this code because it is in the `MouseMove` handler. We return as soon as we possibly can. Checking if a point is in a region is a mathematically complex operation, so you don't want to do that any more times than necessary.

Build and run the application. As you move the mouse over the rectangle, region, and path, it is filled with a color.

Drawing During Mouse Events

The hit testing example also demonstrates the technique of creating and destroying a `Graphics` object during the processing of the `MouseMove` event. This is in contrast to a `Paint` event handler, where a `Graphics` object is passed as part of the `PaintEventArgs` argument. Remember that if you create it, you must dispose of it at the end of the method. Notice that we do our drawing in the event handler itself rather than calling `Invalidate()`.

Drawing during the mouse event can be a little different from drawing during a `Paint` event in this regard. For the most part, when drawing during a mouse event, for performance reasons, you might not want to go through the `Invalidate/Paint` cycle. Primarily, you want to draw directly to the drawing surface, in the mouse event handler itself. Although, having said that, any substantial painting should certainly go into the `Invalidate/Paint` cycle, or you risk losing responsiveness. In the end, there are no hard-and-fast rules, and you should be aware of and assess the suitability of both possible techniques when coding each application.

 Previous

Next 

 Previous

Next 

Mouse Event Routing

When no mouse button is pressed, mouse events are sent to the window that the mouse is over. When a mouse button is pressed, until it is released, the mouse is *captured*. This means that all subsequent `MouseMove` events, until the mouse is released, are sent to the window in which the mouse button went down, even if the mouse cursor moves outside the window. If the mouse moves outside the window, the location of the mouse move events will be outside the window, but they will still go to the window in which the button went down. In addition, the `MouseUp` event will go to the same window, which is very

Often, a custom control will note that the mouse went down in the control. The custom control expects that it will receive the `MouseUp` event so that it can change state, noting that the mouse button is no longer down. However, if the `MouseUp` event went to a different window, the custom control would be left thinking that the mouse button was still down. You can occasionally see this bug in commercial applications, and you don't want to make the same mistake in yours.

There is an exception to this rule. If you were implementing a drag-and-drop operation (but not using the built-in drag-and-drop functionality in Windows Forms) that could drop outside the window in which the mouse button went down, after the `MouseDown` event, you would set the `Capture` property of the `Control` class to `false`. This changes the behavior so that the `MouseMove` events and the `MouseUp` events will go to whatever window the mouse is over. If we want to drag a row from a list in one window, and drop that row in a list in a different window, we want all of the mouse events to go to the window that is currently under the mouse, not the window in which the mouse went down.

Capturing the Mouse

To demonstrate the issues involved with capturing the mouse, and how to address those issues, we will create a small application that emulates the appearance and behavior of a push button. First, let's look at the logic for creating a button:

- If the mouse button goes down outside the button rectangle, we ignore the event.
- If the mouse button goes down inside the button, we set a Boolean variable named `mouseDownInButton` to `true`. In addition, we draw the button with a down appearance, and set another Boolean variable, `buttonDown`, to `true`.
- If the mouse moves outside the button while the mouse is still down, we set `buttonDown` to `false`, and draw the button with an up appearance.
- If the mouse moves back inside the button while the mouse is still down, we set `buttonDown` to `true`, and draw the button with a down appearance.
- If the user releases the mouse outside the button, we take no action apart from setting `mouseDownInButton` to `false`.
- If the user releases the mouse inside the button, we generate an event. We set `mouseDownInButton` to `false`. We draw the button with an up appearance.

This logic points out the problem if the control doesn't receive the `MouseUp` event. In this case, the `mouseDownInButton` Boolean will not be set to `false`, and the control will incorrectly think that it still needs to process the `MouseMove` events.

The next example demonstrates mouse routing. It displays a form on which two rectangles have been drawn, as shown in [Figure 14-5](#).

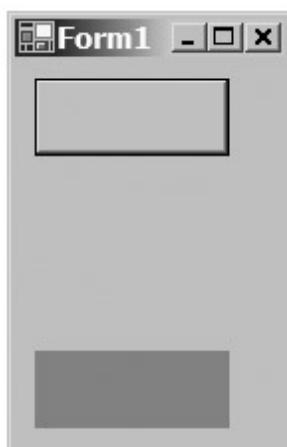


Figure 14-5: Routing mouse events example

The top rectangle is designed to act like a button, which, when clicked, displays a message box saying that it has been clicked. However, instead of using a real button control, we simulate it by drawing the rectangle and handling the required mouse events from the main form class.

If you press the left mouse button over this pseudo-button, you see its sunken appearance, as shown in [Figure 14-6](#). (Perhaps it's not so realistic, but it is enough for our purposes.) The dialog box doesn't appear until the left button is released, completing the "click."

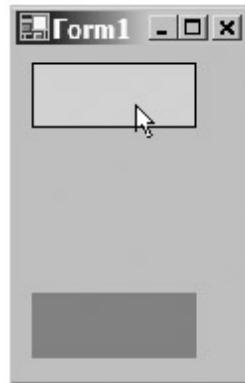


Figure 14-6: Example with pressed button

The lower rectangle is a pretend battery. It pops up a dialog box that says, "Your battery is fully charged," when the mouse moves over it. The rectangle is there to demonstrate a subtle problem with mouse capturing: in some situations, your form can lose the mouse capture. So, let's proceed with building the example.

Create a new Windows Application, call it `RoutingMouseEvents`, and modify the `Form1.cs` module as described here.

First, we need a reference to the `GraphicsOutlineModel` namespace, since we will be using our `GraphicsOM` library (discussed in [Chapter 10](#) and available with the rest of this book's downloadable code).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using GraphicsOutlineModel;
```

Recall that the `GraphicsOM` class contains routines for drawing 3D rectangles, which we will use when drawing our pseudo-button. You will also need to add a reference to the `GraphicsOutlineModel` assembly in the project.

Next, we need to add some fields to the `Form1` class:

```
public partial class Form1 : Form
{
    private Rectangle buttonRect;
    private bool mouseDownInButton;
    private bool buttonDown;
```

The purposes of these fields should be self-explanatory. They store the coordinates of the rectangle where the pseudo-button will be placed, as well as whether the mouse left button has been pressed in the pseudo-button and whether the pseudo-button is down.

Next, we initialize the rectangle that defines our pseudo-button:

```
public Form1()
{
```

```
    InitializeComponent();
```

```
buttonRect = new Rectangle(10, 10, 100, 40);
}
```

Now we add a routine that will draw the pseudo-button. Notice how in the following code we check for a null `Graphics` reference and if so, instantiate a temporary `Graphics` object. The pseudo-button is actually drawn by creating a couple rectangles inside each other using the `GraphicsOM` class to produce the raised or sunken effect, depending on the `up` parameter passed to this method:

```
private void drawButton(Rectangle rect, Graphics g, bool up)
{
    bool disposeGraphics = false;
    if (g == null)
    {
        g = this.CreateGraphics();
        disposeGraphics = true;
    }
    GraphicsOM g2 = new GraphicsOM(g);
    g2.DrawRectangle(Pens.Black, rect);
    Rectangle innerRect = new Rectangle(rect.Location, rect.Size);
    innerRect.Inflate(new Size(-1, -1));
    if (up)
        g2.Draw3DRectangle(innerRect, ThreeDStyle.Raised, 2);
    else
        g2.FillRectangle(Brushes.LightGray, innerRect);
    if (disposeGraphics)
        g.Dispose();
}
```

Next, add the event handlers. First, create a handler for the `Paint` event, which you should wire up to the event in the normal way. This handler simply draws the pseudo-button and the battery rectangle:

```
private void Form1_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    drawButton(buttonRect, g, true);
    // Draw the rectangle that simulates
    // "Your battery is fully charged.
    g.FillRectangle(Brushes.Red, new Rectangle(10, 150, 100, 40));
}
```

The `MouseDown` event handler simply checks whether the mouse button has been pressed in the pseudo-button and, if so, redraws the button in a pressed state and updates the `mouseDownInButton` and `buttonDown` fields accordingly:

```
private void Form1_MouseDown(
    object sender, System.Windows.Forms.MouseEventArgs e)

{
    if (e.Button == MouseButtons.Left && buttonRect.Contains(e.X, e.Y))
    {
        mouseDownInButton = true;
        buttonDown = true;
        drawButton(buttonRect, null, false);
    }
}
```

Similarly the `MouseUp` handler checks if the mouse button had previously been pressed in the pseudo-button; in which case, we know a mouse click is being completed, so we need to display the dialog box and update the appearance of our pseudo-button:

```
private void Form1_MouseUp(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDownInButton && e.Button == MouseButtons.Left)
    {
        drawButton(buttonRect, null, true);
        mouseDownInButton = false;
    }
}
```

```
    if (buttonRect.Contains(e.X, e.Y))
        MessageBox.Show("Button was pressed");
}
}
```

The MouseMove event handler is a bit more complex:

```
private void Form1_MouseMove(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDownInButton)
    {
        if (buttonRect.Contains(e.X, e.Y))
        {
            if (! buttonDown)
            {
                drawButton(buttonRect, null, false);
                buttonDown = true;
            }
        }
        else
        {
            if (buttonDown)
            {
                drawButton(buttonRect, null, true);
                buttonDown = false;
            }
        }
    }

    if (new Rectangle(10, 150, 100, 40).Contains(new Point(e.X, e.Y)))
        MessageBox.Show("Your battery is fully charged.");
}
}
```

In this handler, we first check whether we are in the middle of a mouse click; in other words, the mouse button had been pressed while in our pseudo-button on the form. If so, then we need to redraw the button correctly, according to whether the user has moved the mouse outside its area. Finally, we check if we are in the battery area, and if so, display the appropriate message box.

Handling the CaptureChanged Event

The application we just created works great almost all of the time. We can press the button with the left mouse button, drag the mouse outside the window, and release the button outside the window. Since the mouse was captured, the `MouseUp` event went to our `Form1_MouseUp` event handler, which changed state appropriately by setting the `mouseDownInButton` variable to `false`.

However, if another window is put up, perhaps not even by our application, the `MouseUp` event will go to that window, even though the mouse is captured. If you move the mouse over the red rectangle in the form, the example puts up a modal dialog box, simulating what would happen if such a dialog box were put up by another application. To see an example of this bug, press the left mouse button in the pseudo-button, and without releasing the mouse button, drag the mouse over the red rectangle. A message box is put up. Dismiss the message box (using the Enter key, for example), and then drag the mouse back over the pseudo-button. Since the `MouseUp` event went to the message box, our pseudo-button still thinks that it has not received a `MouseUp` event, and behaves as though the mouse were down.

Another problem can be caused by the user doing something, albeit something quite unusual. Press the left mouse button in the pseudo-button. Now press and release the right mouse button, while continuing to hold down the left mouse button. Next, drag the mouse cursor outside the window, and then release the left mouse button. If you now drag the mouse back over the pseudo-button, you can see that the pseudo-button thinks that the left mouse button is still down, even though it is not.

What happened is that pressing and releasing the right mouse button caused the mouse to be no longer captured. When you released the left mouse button while outside the window, the `MouseUp` event went to whatever window the mouse was over, and the pseudo-button didn't receive it.

There is a solution, which is to watch for the `WM_CAPTURECHANGED` event, which is a Win32 message.

Windows raises this message whenever the capture state of the mouse changes, no matter how it changes. However, we need to dip down into the Win32 API to watch for this event, since Windows Forms don't define a corresponding .NET event.

We need to make the following additions to the code for the `MouseRoutingEvent` form. First, add a delegate definition inside the `Form1` class. This will define the event handler for when a `WM_CAPTURECHANGED` message occurs.

```
public delegate void CaptureChanged();
```

Now we define an inner class, again inside the `Form1` class, to handle the low-level Windows procedure that will respond to the `WM_CAPTURECHANGED` message:

```
class CaptureChangedWindow : NativeWindow
{
    public CaptureChanged OnCaptureChanged;

    protected override void WndProc(ref Message m)
    {
        if (m.Msg == 533) // WM_CAPTURECHANGED
            OnCaptureChanged();
        base.WndProc(ref m);
    }
}
```

Inside the Windows API, windows are represented by numbers known as *window handles*. Normally, this is all dealt with internally by the `Form` class. However, if you want to work at a low level, Microsoft makes available the `System.Windows.Forms.NativeWindow` class. This has none of the extensive windowing features of the `Form` class, but it does wrap a native Windows handle and provides a virtual method, `WndProc()`, that represents the Windows procedure called when messages are received. Hence, we define our `CaptureChangedWindow` class to derive from `NativeWindow`, and override `NativeWindow.WndProc()` to provide a handler for the `WM_CAPTURECHANGED` message (which has a value of 533). Our override of this method simply invokes the `CaptureChanged` delegate.

Next, we declare a member field to hold a `CaptureChangedWindow` instance:

```
public class Form1 : System.Windows.Forms.Form
{
    public delegate void CaptureChanged();
    CaptureChangedWindow ccw;
```

Now we define a method in `Form1` that will be invoked when the `CaptureChanged` delegate is invoked. This is what we do when the form detects a capture change: set `mouseDownInButton` to `false` to indicate we are no longer in a potential mouse-click event.

```
private void CaptureChangedEventHandler()
{
    mouseDownInButton = false;
}
```

Finally, we need to modify the `Form1` constructor to create and initialize the nested class:

```
public Form1()
{
    InitializeComponent();
    buttonRect = new Rectangle(10, 10, 100, 40);

    ccw = new CaptureChangedWindow();
    ccw.AssignHandle(Handle);
    ccw.OnCaptureChanged +=
        new CaptureChanged(CaptureChangedEventHandler);

}
```

The following line initializes `ccw` with the same window handle that our `Form1` instance contains:

```
ccw.AssignHandle(Handle);
```

This means that messages sent to `Form1` that would normally cause events to be raised will additionally be passed to `ccw` for processing in our `WndProc()` override.

That's all we need to do. Now the application behaves correctly, regardless of whether you press and release the right mouse button while the left mouse button is held down, or whether you pass the mouse cursor over the rectangle that simulates a modal dialog opening while the mouse is captured.

Note If you press and release the right mouse button while the left mouse button is held down, the pseudo-button will remain pressed, no matter where you move the mouse, even if you release both mouse buttons. Although this behavior is unusual, the circumstances that cause it are exceptional. It is peculiar to Windows Forms controls and not exhibited by pre-.NET code. It appears to be linked to the architecture of Windows Forms event handling, and there is no easy way to avoid it.

The technique of noting when the capture window changed is primarily of interest when implementing dragging events. Otherwise, the default behavior of the .NET Framework is fine. However, as soon as your application's users will be dragging something with the mouse, you need to pay attention to the structure of your window. You need to program your application in such a way that you know which window the mouse-up event will be routed to.

We are now going to leave mouse events behind for a while and take a look at GDI. The reason is that our next example, which will demonstrate dragging some text around the screen with the mouse, will need to use a couple of functions from the old GDI API.

 Previous

Next 

 Previous

Next 

GDI and BitBlt

Before we get to the next topic about mouse-event handling, I need to introduce the `BitBlt()` method, which comes from the old GDI API. GDI was the forerunner of GDI+. It serves the same purpose, allowing you to draw graphics and manipulate bitmaps, but it works at a lower level and is not .NET-aware. It consists of a large number of C-style functions, and if you want to call them from C# or Visual Basic .NET code, you need to use the `DllImport` attribute and the `PInvoke` mechanism.

WHY USE GDI?

The main reason for using GDI in some situations is that there are some things you can do with GDI that you can't do with GDI+.

First, GDI+ is a very high-level class library. This makes it very quick for you to write basic drawing code, but it has the disadvantage that sometimes, if you want to do something unusual, you might find that there is simply no way to perform the task in GDI+. A case in point, and the real reason why we will be using GDI in the next examples, is that the GDI+ classes do not allow you to copy data from the screen to an image contained in memory. There is no fundamental reason why you shouldn't be able to do that; it's just the GDI+ classes have been written in a way that doesn't support BitBltting from the screen. If you use GDI though, you can do so.

When you call the `BitBlt()` function, you are required to separately supply the device contexts (drawing surfaces) for the source and destination surfaces. By contrast, although `Graphics.DrawImage()` does a `BitBlt()` internally, the destination device context is defined as being the one represented by the `Graphics` object you are using. The source device context will be one that is internally created in GDI+ specifically to represent the image that you are copying, and will therefore necessarily be a memory device context. This extra flexibility can sometimes give GDI the edge and make new techniques available to the developer.

You may also find that in a few cases, using GDI might give you a performance edge, because working at a lower level gives you more control over what the computer is doing. Again, `Graphics.DrawImage()` provides an example of this. The internal implementation of this method is undocumented by Microsoft, but there is at least a possibility that each call causes a new device context to be temporarily created. Using GDI, if you are repeatedly drawing from the same bitmap, you would be able to cache the device context.

On the other hand, you should take care with using GDI for performance. Unless there is a clear

optimization that you can perform using GDI, you may find GDI+ actually performs better. That's because, as a newer API, it is likely to be better supported by Microsoft in the future, and indeed, Microsoft is already claiming that GDI+ can offer better performance than GDI. Also, GDI+ has some newer features not available in GDI (because when GDI was written, there was no possibility of any hardware supporting those features). An example of this is gradient brushes defined by paths. In general, my advice is to stick to using GDI+ unless, in a particular situation, there is a clear reason for resorting to GDI.

Introducing BitBlt

BitBlt is often pronounced "BitBlit" and stands for *Bitmap Block Transfer*, which is the way that many images are copied to the screen. A bitmap, or a rectangular area within a bitmap, is copied across from one drawing surface to another using support from the graphics card hardware, which means the operation occurs very quickly. A BitBlt is what actually happens internally in the `Graphics.DrawImage()` method. A drawing surface can exist in the computer's memory or on the screen, or can represent a printer, and BitBlt operations can be used equally well to transfer data between any of these types of surfaces. Clearly, the process of drawing to the screen normally means performing a BitBlt from the computer's memory to the screen memory.

Each drawing surface is represented in Windows by a data structure known as a *device context*. The device context contains all the information needed to tell Windows how to draw to that surface, as well as a reference to where the memory containing the data for all the pixels in that surface is. The `Graphics` object, which you have been using routinely, is actually a .NET wrapper class around a device context.

Up to now, we have viewed BitBlt as a simple case of copying rectangular areas. In fact, you can do quite a bit more with the `BitBlt()` method than just copying pixels from one location to another. In addition, it is possible to set each destination pixel to some value based on combining the source pixel and the original color in the destination pixel, which offers possibilities for a variety of interesting effects.

`WinGdi.h` defines the constants that define these effects as shown in [Table 14-3](#).

Table 14-3: BitBlt Operations

Operation	Hex Code	Description
SRCCOPY	0x00CC0020	dest = source
SRCPAINT	0x00EE0086	dest = source OR dest
SRCAND	0x008800C6	dest = source AND dest
SRCINVERT	0x00660046	dest = source XOR dest
SRCCERASE	0x00440328	dest = source AND (NOT dest)
NOTSRCCOPY	0x00330008	dest = (NOT source)
NOTSRCCERASE	0x001100A6	dest = (NOT source) AND (NOT dest)
MERGECOPY	0x00C000CA	dest = (source AND pattern)
MERGEPAINT	0x00BB0226	dest = (NOT source) OR dest
PATCOPY	0x00F00021	dest = pattern
PATPAINT	0x00FB0A09	dest = DPSnoo
PATINVERT	0x005A0049	dest = pattern XOR dest
DSTINVERT	0x00550009	dest = (NOT dest)
BLACKNESS	0x00000042	dest = BLACK
WHITENESS	0x00FF0062	dest = WHITE

In [Table 14-3](#), the AND, OR, and NOT operations are bitwise operations performed on the RGB values that describe the color at each pixel. If you are using GDI+ and `DrawImage()`, these raster operation flags are not available; you can copy only pixels. In this book's examples, we will use only the SRCCOPY

operation. As mentioned earlier, our main reason for dropping to GDI is so that we can transfer data from the screen to memory.

Using BitBlt()

The technique to use `BitBlt()` involves the following:

- Declare a `DllImport` attribute to specify the DLL location that contains the implementation of the `BitBlt()` method.
- Declare the external `BitBlt()` method. This consists of the signature of the method, including its return value, and the types of all of its arguments.
- Get a handle to the device context (`Hdc`) for the drawing surface. To get the handle, you call the `GetHdc()` method of the `Graphics` object for your drawing surface.
- Call the `BitBlt()` method, passing the `Hdc` as an argument.
- After you are finished with the `Hdc`, remember to release the handle to the device context. You release it by calling the `ReleaseHdc()` method of the `Graphics` class.

Note that if you do not have an existing `Graphics` object, you can use the GDI function `CreateDC()` to obtain one. In that case, you will need to call another GDI function, `ReleaseDC()`, to indicate when you no longer need the device context in order to release the associated resources. Failure to do so won't just cause memory leaks; it can theoretically cause future drawing operations to crash, because there are restrictions on the number of device contexts that can exist simultaneously (although precise details of this are not documented). As long as you release device contexts when you're finished with them, you'll be fine.

The following is a very simple example that uses the `BitBlt()` function. Its sole purpose is to demonstrate the use of the `BitBlt()` method, before we go on to use it properly in the next mouse event-handling example. In this example, we will draw a small figure in a form, and then `BitBlt` it to another place in the window.

Create a new Visual C# Windows Application, calling it `BitBltExample`. Add a constant to the `Form1` class as a mnemonic for the `SRCCOPY` raster flag:

```
public partial class Form1 : Form
{
    const int SRCCOPY = 0xcc0020;
```

Add the `DllImport` attribute and the declaration of the external `BitBlt()` method to the `Form1` class, as follows:

```
[System.Runtime.InteropServices.DllImportAttribute("gdi32.dll")]
private static extern int BitBlt(
    IntPtr hdcDest,          // handle to destination DC (device context)
    int nXDest,              // x-coord of destination upper-left corner
    int nYDest,              // y-coord of destination upper-left corner
    int nWidth,              // width of destination rectangle
    int nHeight,             // height of destination rectangle
    IntPtr hdcSrc,            // handle to source DC
    int nXSrc,                // x-coordinate of source upper-left corner
    int nYSrc,                // y-coordinate of source upper-left corner
    System.Int32 dwRop        // raster operation code
);
```

Add a `Paint` event, and modify it as follows:

```
private void Form1_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.White, ClientRectangle);
    g.DrawRectangle(Pens.Black, 10, 10, 40, 40);
    IntPtr dc = g.GetHdc();
    BitBlt(dc, 70, 0, 60, 60, dc, 0, 0, SRCCOPY);
```

```
// WinGdi.H contains constants for BitBlt  
// 0xCC0020 is the SRCCOPY raster operation  
g.ReleaseHdc(dc);  
}
```

When you run the example, it appears as shown in [Figure 14-7](#).

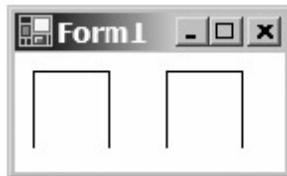


Figure 14-7: A BitBlted rectangle

Now that you've seen how BitBlt works, let's see its application in drawing during mouse events.

[Previous](#)

[Next](#)

[Previous](#)

[Next](#)

Drawing While Dragging

When your application's users are dragging and dropping some object from one part of your application to another, you may want to show what they are dragging. In this section, we will develop an example that illustrates one way to implement this feature.

This example will use the GDI `BitBlt()` function introduced in the [previous section](#). This is necessary because we want to copy a portion of the display to a backing store—a bitmap held in memory. This technique consists of using the `BitBlt()` functionality as follows:

- We get a device context for the display.
- From the device context, we create a `Graphics` object, so that we can use GDI+ to draw directly to the display.
- We then use `BitBlt()` function to save the contents under the cursor to a backing store.
- After we have saved a region of the display to a backing store, we can draw directly to the display, knowing that we can repair it using the backing store.

This example draws using an *alpha* technique, so that what the user is dragging is semi-transparent.

To begin, create a new Windows Application and call it `AlphaDragExample`. This example will use unsafe code blocks. This means that it won't compile unless you tell Visual Studio .NET to allow unsafe code. So, let's see how to do that first.

Compiling with Unsafe Code

Unsafe code is different from unmanaged code. Managed code versus unmanaged code has to do with the issue of running with the CLR runtime, which manages such things as security and the garbage collector. Unsafe code is code that does things such as pointer arithmetic or calling into unmanaged code. The unsafe code in this example calls into unmanaged code, so it needs to be declared as unsafe code.

You modify a project so that it can compile when it contains unsafe code via the Project Properties window. Select Build in the tabs to the left of the Properties window, and make sure that the box at the top of the dialog box shows All Configurations. Then locate the Allow Unsafe Code option and put a checkmark in its box, as shown in [Figure 14-8](#).

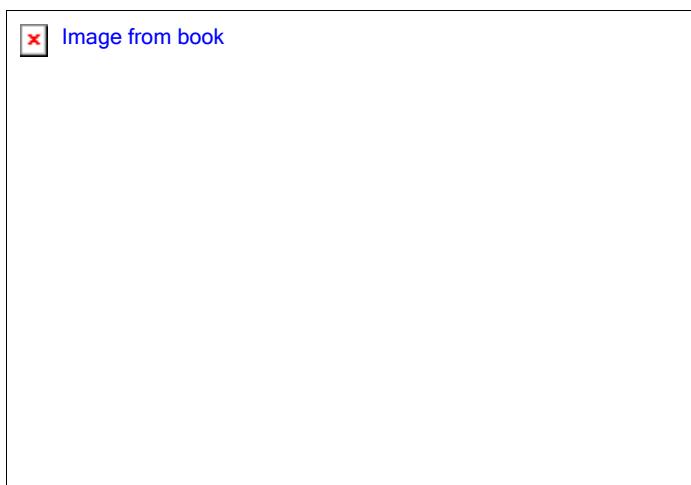


Figure 14-8: Allowing unsafe code

Coding Drawing While Dragging

To start off the code for the `AlphaDragExample` application, add a couple of namespace references:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
using System.Runtime.InteropServices;
```

We need `System.Drawing.Drawing2D` because we will be using `LinearGradientBrush`, and `System.Runtime.InteropServices` to access the `DllImport` attribute.

Next, inside the `Form1` class, add the following `DllImport` definitions:

```
[DllImport("gdi32.dll")]
private static extern bool BitBlt(
    IntPtr hdcDest,          // handle to destination DC (device context)
    int nXDest,              // x-coord of destination upper-left corner
    int nYDest,              // y-coord of destination upper-left corner
    int nWidth,              // width of destination rectangle

    int nHeight,             // height of destination rectangle
    IntPtr hdcSrc,            // handle to source DC
    int nXSrc,                // x-coordinate of source upper-left corner
    int nYSrc,                // y-coordinate of source upper-left corner
    System.Int32 dwRop        // raster operation code
);

[DllImport("gdi32.dll")]
private static extern IntPtr CreateDC(
    String driverName,
    String deviceName,
    String output,
    IntPtr lpInitData
);

[System.Runtime.InteropServices.DllImportAttribute("gdi32.dll")]
private static extern bool DeleteDC(
    IntPtr dc
);
```

```
[System.Runtime.InteropServices.DllImportAttribute("user32.dll")]
private static extern unsafe bool ClientToScreen(
    IntPtr hWnd,           // handle to window
    Point* lpPoint         // screen coordinates
);
```

We will also need the following fields in our Form1 class:

```
const int SRCCOPY = 0xcc0020;
private Point dragMePosition;
private bool mouseDown = false;
private Point mouseDelta;
private Bitmap backingStore;
private Point backingStorePos;
private String dragString;
private Graphics backingStoreGraphics;
private IntPtr backingStoreDC;
private Graphics displayGraphics;
private IntPtr displayDC;
private Size dragImageSize;
private Font dragStringFont;
```

The purposes of these fields should be clear from their names. The backing store is the bitmap that will be stored behind the scenes containing the image to be dragged, and there are various fields describing its size and contents, as well as a device context handle, represented in .NET by an IntPtr instance.

Initialize several of these fields in the constructor:

```
public Form1()
{
    InitializeComponent();
    dragString = "Drag Me!";
    dragMePosition = new Point(100, 100);
    dragStringFont = new Font("Times New Roman", 14, FontStyle.Bold);
}
```

Some initialization also takes place in the Load event handler, which you should add to the project. We calculate the size of the string to be dragged there rather than in the constructor, since we need to be able to access a Graphics object:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Graphics g = this.CreateGraphics();
    SizeF temp = g.MeasureString(this.dragString, this.dragStringFont);
    dragImageSize = new Size((int)temp.Width, (int)temp.Height);
    g.Dispose();
}
```

Next, add a couple of helper methods to work with the GDI objects we will be using:

```
private void getDisplayDC()
{
    displayDC = CreateDC("DISPLAY", null, null, (System.IntPtr)null);
    displayGraphics = Graphics.FromHdc(displayDC);
}

private void disposeDisplayDC()
{
    displayGraphics.Dispose();
    DeleteDC(displayDC);
}

private unsafe Point controlToScreen(int x, int y)
{
    Point p = new Point(x, y);
    Point* pp = &p;
    ClientToScreen(Handle, pp);
    return p;
}
```

}

As you can see from this code, to get a device context for the display, and then to create a `Graphics` object, we do the following:

```
displayDC = CreateDC("DISPLAY", null, null, (System.IntPtr)null);
displayGraphics = Graphics.FromHdc(displayDC);
```

The device context is removed by calling `DeleteDC()`.

The `controlToScreen()` method that we define uses the API function `ClientToScreen()`, which is GDI's way of converting from client coordinates to screen coordinates. The function requires a pointer to a `Point` object, and we need to use unsafe code to get such a pointer.

Now let's examine the `Paint` event handler:

```
private void Form1_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Bisque, ClientRectangle);
    Font bf = new Font("Times New Roman", 32);
    g.DrawString("Bonjour Le Monde", bf, Brushes.Blue, 10, 10);
    bf.Dispose();
    drawDragImage(
        g, false, dragMePosition.X, dragMePosition.X, dragString);
}
```

This handler simply draws the `Bonjour Le Monde` text, and then calls another method, `drawDragImage()`, which draws the `Drag Me` text in its normal position. `drawDragImage()` looks like this:

```
private void drawDragImage(
    Graphics g, bool alphablend, int x, int y, String s)
{
    Rectangle r = new Rectangle(
        x, y, dragImageSize.Width, dragImageSize.Height);

    // Draw background
    if (alphablend)
    {
        Color c1 = Color.FromArgb(0, Color.Red);
        Color c2 = Color.FromArgb(80, Color.Red);
        Rectangle r1 = new Rectangle(r.Left, r.Top, r.Width / 2, r.Height);
        Rectangle r2 = new Rectangle(
            r.Left + r.Width / 2, r.Top, r.Width / 2, r.Height);
        LinearGradientBrush lgb1 = new LinearGradientBrush(r1, c1, c2, 0f);
        LinearGradientBrush lgb2 = new LinearGradientBrush(r2, c2, c1, 0f);
        g.FillRectangle(
            lgb1, new Rectangle(r1.Left + 1, r1.Top, r1.Width + 1, r1.Height));
        g.FillRectangle(lgb2, r2);
        lgb1.Dispose();
        lgb2.Dispose();
    }
    else
        g.FillRectangle(Brushes.Red, r);

    // Draw the text
    Color c;
    if (alphablend)

        c = Color.FromArgb(80, Color.Black);
    else
        c = Color.Black;
    Brush blackBrush = new SolidBrush(c);
    g.DrawString(s, this.dragStringFont, blackBrush, r);
    blackBrush.Dispose();
```

}

This method is more complex because not only does it handle drawing the image in its normal location, but it also draws semitransparently when the alphablend parameter is set to `true`. If `alphablend` is `true`, we construct a background rectangle using linear gradient brushes with colors of varying opacity; otherwise, we just construct a solid red brush to paint the text. Either way, this method basically constructs the various brushes needed and draws the text.

Note that `Color.FromArgb()`, when passed an integer and a `Color`, gives a color with the color itself determined by the `Color` parameter and the opacity determined by the integer. A value of 0 gives a completely transparent image; 255 is completely opaque. Intermediate values will draw the image, but with the background showing through to some extent. So, the following code will store a semitransparent Red color in the local variable `c2`:

```
Color c2 = Color.FromArgb(80, Color.Red);
```

Now let's look at the three mouse-event handlers. We need handlers for the `MouseDown`, `MouseMove`, and `MouseUp` events. We will start with the code for `MouseDown`:

```
private void Form1_MouseDown(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    Rectangle dragMeRectangle = new Rectangle(
        dragMePosition, new Size(dragImageSize.Width, dragImageSize.Height));
    if (dragMeRectangle.Contains(new Point(e.X, e.Y)))
    {
        mouseDown = true;
        mouseDelta = new Point(
            e.X - dragMePosition.X, e.Y - dragMePosition.Y);

        Point screenPoint = controlToScreen(e.X, e.Y);
        screenPoint.X -= mouseDelta.X;
        screenPoint.Y -= mouseDelta.Y;
        getDisplayDC();

        backingStore =
            new Bitmap(dragImageSize.Width, dragImageSize.Height);
        backingStoreGraphics = Graphics.FromImage(backingStore);
        backingStoreDC = backingStoreGraphics.GetHdc();
        BitBlt(backingStoreDC, 0, 0, backingStore.Width,
            backingStore.Height, displayDC, screenPoint.X,
            screenPoint.Y, SRCCOPY);
        backingStorePos = new Point(screenPoint.X, screenPoint.Y);

        drawDragImage(
            displayGraphics, true, screenPoint.X, screenPoint.Y, dragString);
    }
}
```

We don't actually need to do anything unless the user has pressed the mouse button inside the text that can be dragged, so that's the first check we make. If that is the case, we record the fact that we are now dragging, as well as a field called `mouseDelta`—this is the starting location of the mouse cursor relative to the top-left corner of the region that can be dragged. We will need this value to work out how far the mouse has moved and therefore where to draw the dragged image while dragging. The bulk of the remainder of this method is concerned with saving the contents of the screen in the dragging rectangle to a memory bitmap—the so-called backing store. Note that we need to use `BitBlt()` to do this, although we are able to stick with GDI+ for the remaining code.

Now let's examine the `MouseMove` event handler:

```
private void Form1_MouseMove(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDown)
    {
        Point screenPoint = controlToScreen(e.X, e.Y);
        screenPoint.X -= mouseDelta.X;
        screenPoint.Y -= mouseDelta.Y;
```

```
    BitBlt(displayDC, backingStorePos.X, backingStorePos.Y,
           backingStore.Width, backingStore.Height, backingStoreDC,
           0, 0, SRCCOPY);
    BitBlt(backingStoreDC, 0, 0, backingStore.Width,
           backingStore.Height, displayDC, screenPoint.X,
           screenPoint.Y, SRCCOPY);
    backingStorePos = new Point(screenPoint.X, screenPoint.Y);
    drawDragImage(
        displayGraphics, true, screenPoint.X, screenPoint.Y, dragString);
}
}
```

This handler doesn't need to do anything unless the `mouseDown` field indicates that we are in the middle of a drag operation. If we are, the screen requires updating. The two `BitBlt()` operations are involved in restoring the screen area where the semitransparent image had been before the mouse moved, and then backing up the screen again. Then we call `drawDragImage()` to actually draw the dragged image in the new position.

The final method we need to consider is the event handler for the `MouseUp` event that ends the drag operation:

```
private void Form1_MouseUp(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDown)
    {
        mouseDown = false;

        BitBlt(displayDC, backingStorePos.X, backingStorePos.Y,
               backingStore.Width, backingStore.Height, backingStoreDC,
               0, 0, SRCCOPY);

        disposeDisplayDC();

        backingStoreGraphics.ReleaseHdc(backingStoreDC);
        backingStoreGraphics.Dispose();

        backingStore.Dispose();
        backingStore = null;
    }
}
```

This handler simply uses a `BitBlt()` to remove the dragged image from the screen. Then, it releases the temporary `Graphics` object and related `Hdc` (hardware device context) that we created on the `MouseDown` event for the drag operation.

This example just shows the bare basics of this technique. A more generalized version would be data-driven, could drag a variety of objects, and so on. However, there would not be anything technically different about a more generalized technique.

Using this technique, you can give your application a sophisticated look, as well as provide a greater level of feedback to the user.

Improving Drawing Performance

If you run the sample application, you will find that it works well, though there may be some flickering as you drag the control. Some flickering can be hard to avoid when you are dealing with moving graphical images. Here are a few suggestions for improving drawing performance:

- Remove any unnecessary work from the drawing routines. In the `AlphaDragExample`, there is some scope for that. For example, the `drawDragImage()` method works out the gradient brushes each time it is called. It would be more efficient to work out these brushes once and cache them.
- The `drawDragString()` method in the example uses `Graphics.DrawString()` to draw the text. In general, drawing text is exceptionally slow. If the same text needs to be drawn multiple times, consider drawing it once to an in-memory bitmap, then `BitBlting` the bitmap instead whenever you

- If you are using GDI+, examine your code carefully for cases where you might be able to take advantage of GDI's lower-level control to arrange the code more efficiently.
- Consider using a back buffer. .NET forms can use back buffers automatically. Look up the `ControlStyles.DoubleBuffer` enumeration in the MSDN documentation for more details on this.

Previous

Next

Previous

Next

Drag-Scrolling

Sometimes, you want to implement scrolling behavior when the user selects something in the control, and then drags the mouse outside the control. This is the behavior that you see in a word processor when you select text and drag the mouse below the window in which you are editing text. The text in the window starts scrolling as it continues selection. You can also see this behavior in web browsers, image editors, and many other types of applications. I call this action *drag-scrolling* (which is not a standard Windows term).

Related to drag-scrolling is a bug most often found with a custom control that implements some form of vertical or horizontal scrolling. In its crudest form, this bug shows up when dragging and selecting text. When the user drags the mouse outside the client area of the custom control, the control should start scrolling as it continues to select text. However, with some controls or applications, when the mouse moves outside the client area, the client area scrolls, and then stops. When the user moves the mouse again, the client area scrolls again, until the user stops moving the mouse. In order to scroll continuously, the user needs to "wiggle" the mouse below or above the client area while holding down the mouse button. I call this the "wiggle-the-mouse" bug.

Understanding the Wiggle-The-Mouse Bug

To understand why the wiggle-the-mouse bug happens, we need to look at the algorithm used to implement scrolling while selecting. The algorithm consists of the following points:

- Initially, as the user drags within the client area of the control, text is selected in the control.
- When the user drags below or above the control, the `MouseMove` event handler scrolls the control, and then selects the appropriate text.
- The quantity to scroll is best calculated based on the distance of the mouse above or below the client area of the custom control.
- When implemented incorrectly, the control scrolls with each `MouseMove` event. This is the source of the bug.

The solution is to start a timer that generates mouse events, while the mouse is captured, at regular intervals. You then can scroll with each `MouseMove` event, and scrolling will be continuous even if you don't wiggle the mouse below the custom control. However, this can lead to a variation of the bug. If you trigger the scrolling on the `MouseMove` events that come from the mouse *and* on the `MouseMove` events that you generate from the timer, you will end up with the bug that I call the "wiggle-the-mouse-to-scroll-faster" bug. In other words, wiggling the mouse makes the scroll happen faster, contrary to user expectations. The solution to this is that when automatically scrolling, you only scroll based on the timer-generated `MouseMove` events. However, you can't completely ignore the mouse generated `MouseMove` events. You need to note the location of the mouse and use that location in the generation of the timer-based `MouseMove` events.

To demonstrate the wiggle-the-mouse bug, we'll work through an example. This example consists of a custom control, similar to a `ListBox` control, which allows you to select rows with the mouse, as shown in [Figure 14-9](#).

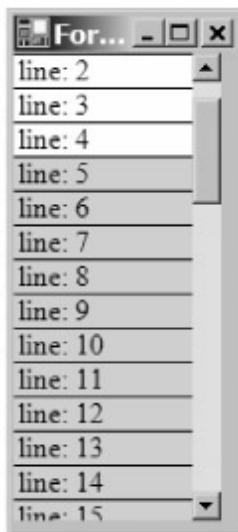


Figure 14-9: Scroll and select list

Although the contents of the control look a bit like a ListBox, it isn't a ListBox. The effect is created entirely by custom drawing code.

The control allows you to select rows with the mouse. While doing this, you can drag above or below the custom control, and the control will scroll as it continues to select rows. The farther above or below that you drag the mouse while scrolling, the faster the custom control scrolls. In the [next section](#), you will see how to fix the bug.

This custom control is a variation on the smooth scrolling custom control presented in the [previous chapter](#), so create a new Windows Control Library project and call it `ScrollAndSelectList`. Rename the C# class to `ScrollAndSelectList.cs` and, similarly, change its `(Name)` property as well. We need to modify the namespaces used in this example, at the top of the code file. This control will also be using the `GraphicsOM` assembly, so add a namespace reference and a reference to the project:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GraphicsOutlineModel;
```

In this case, we can leave the control as derived from `UserControl`. Use the Toolbox to add a `VScrollBar` to it, call it `vScrollBar`, and set its `Dock` property to `Right`.

Now add the following fields to the class:

```
private Point originOfImage = new Point(0, 0);
private Size virSpaceSize = new Size(100, 1000);
private Bitmap bitmap;
private int startingSelectedRow = -1;
private int endingSelectedRow = -1;
private int lineSpacing;

private int nbrRows;
private bool mouseDown = false;
private int draggingStartedRow;
private int old_startingSelectedRow = -1;
private int old_endingSelectedRow = -1;
```

Also add a line to the constructor for the `Form1` class:

```
public ScrollAndSelectList()
{
```

```

    InitializeComponent();

    this.SetStyle(ControlStyles.Opaque, true);
}
}

```

Setting the control style to Opaque prevents the background from being repainted when the control is painted. This is important to prevent flickering while you are dragging the mouse to select rows.

Now we will do some housekeeping. The control will work by drawing from a large bitmap the size of the virtual space; that is, big enough to display all rows in the pseudo-list box. The technique we will use is to just display the appropriate portion of this bitmap on the screen using `Graphics.DrawImage()`. The size of this space is accessed via the `VirtualSpaceSize` property:

```

public Size VirtualSpaceSize
{
    get
    {
        return virSpaceSize;
    }
    set
    {
        // Adjust size so that there is an integral number of rows.
        lineSpacing = (int)Font.Height;
        virSpaceSize = value;
        nbrRows = virSpaceSize.Height / lineSpacing;
        virSpaceSize.Height = nbrRows * lineSpacing + 1;
        bitmap = new Bitmap(
            this.virSpaceSize.Width, this.virSpaceSize.Height);
        drawIntoImage();
        Invalidate();
        adjustScrollBars();
    }
}
}

```

As you can see, setting this property causes a new bitmap to be allocated, followed by calls to a couple of methods that we will code soon. `drawIntoImage()` actually draws the list box into the bitmap. Then, we call `Invalidate()` to raise a `Paint` event and get the bitmap drawn to the screen. Finally, `adjustScrollBars()` sets up the scroll bars based on the size of the virtual space.

We also need to make sure the bitmap resource is cleaned up, so we modify the `Dispose()` call:

```

protected override void Dispose(bool disposing)
{
    bitmap.Dispose();
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
}

```

The `Dispose` method is in the `ScrollAndSelectList.Designer.cs`. Ordinarily, you can't see this file. However, if you click the Show All Files button in the toolbar, you can expand a node in the Solution Explorer and see the file.

Next, we define a property that gives us the actual drawing area; that is, the area of the control that is not occupied by the scroll bar:

```

private Rectangle ScrollingImageRectangles
{
    get
    {
        return new Rectangle(0, 0, this.vScrollBar.Left,
            this.Height);
    }
}

```

The `adjustScrollBars()` method sets up the scroll bars with the data they need to display correctly,

given the current position of the control we are looking at:

```
private void adjustScrollBars()
{
    vScrollBar.Minimum = 0;
    vScrollBar.Maximum = this.virSpaceSize.Height - 2;
    vScrollBar.LargeChange = ScrollingImageRectangle.Height - 1;
    vScrollBar.SmallChange = lineSpacing;
    vScrollBar.Value = originOfImage.Y;
}
```

The `drawIntoImage()` method actually draws the bitmap we will be displaying. It simply iterates through the rows, writing out some text in each one, as well as drawing the lines that separate the rows:

```
private void drawIntoImage()
{
    int y;

    Graphics g = Graphics.FromImage(bitmap);
    g.FillRectangle(Brushes.White, new Rectangle(
        0, 0, bitmap.Width, bitmap.Height));
    GraphicsOM g2 = new GraphicsOM(g);

    for (int i = 0; i < nbrRows; ++i)
    {
        y = i * lineSpacing;
        g2.DrawLine(Pens.Black, 0, y, virSpaceSize.Width, y);
        Rectangle r = new Rectangle(
            0, y, virSpaceSize.Width, lineSpacing - 1);
        if (i >= startingSelectedRow && i <= endingSelectedRow)
            g2.FillRectangle(Brushes.LightBlue, new Rectangle(
                r.Left, r.Top + 1, r.Width, r.Height));
        g.DrawString("line: " + i, Font, Brushes.Black, r);
    }
    y = lineSpacing * nbrRows;
    g2.DrawLine(Pens.Black, 0, y, virSpaceSize.Width, y);
    g.Dispose();
}
```

Now let's consider the `Paint` event handler. This creates the backing bitmap, if necessary, before drawing using a helper method, `drawImage()`. We will consider both methods together:

```
private void ScrollAndSelectList_Paint(
    object sender, System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    if (bitmap != null)
    {
        lineSpacing = (int)Font.Height;
        nbrRows = virSpaceSize.Height / lineSpacing;
        drawIntoImage();
        drawImage(g, this.originOfImage.X, this.originOfImage.Y);
    }
    else
    {
        lineSpacing = (int)Font.Height;
        nbrRows = virSpaceSize.Height / lineSpacing;
        bitmap = new Bitmap(virSpaceSize.Width,
            virSpaceSize.Height);
        drawIntoImage();
        adjustScrollBars();
    }
}

private void drawImage(Graphics g, int x, int y)
{
    if (bitmap != null)
```

```
        g.SetClip(ScrollingImageRectangle);
        Rectangle drawRect = new Rectangle(new Point(x, y),
            ScrollingImageRectangle.Size);
        g.DrawImage(bitmap, ScrollingImageRectangle,
                    drawRect, GraphicsUnit.Pixel);
    }
}
```

One more housekeeping method, `getRowFromHitTest()`, examines the mouse position and determines what row this is in. Since this method will be called by the mouse event handlers we write, it takes the mouse position from a `MouseEventArgs` parameter:

```
private int getRowFromHitTest(System.Windows.Forms.MouseEventArgs e)
{
    Point newPoint = new Point(e.X, e.Y + originOfImage.Y);
    int row = newPoint.Y / lineSpacing;
    return row;
}
```

Now we will get on to the part of the code relating to scroll-selecting. First, the event handler for `MouseDown`. This code simply checks in which row the mouse was pressed, sets some variables to indicate we are now in a drag-select operation, and redraws the current row selected:

```
private void ScrollAndSelectList_MouseDown(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    int row = getRowFromHitTest(e);
    this.startingSelectedRow = row;
    this.endingSelectedRow = row;
    draggingStartedRow = row;
    drawIntoImage();
    this.Invalidate();
    mouseDown = true;
}
```

The next stage in drag-selecting is when the user moves the mouse. Here's the event handler for the `MouseMove` event:

```
private void ScrollAndSelectList_MouseMove(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    processMouseMove(e);
}
```

As you can see, the `MouseMove` handling has been placed in a separate method, `processMouseMove()`. At the moment, that extra method call looks redundant, but it will come in handy later when we fix the wiggle-the-mouse bug, which will involve adding some more code to `ScrollAndSelectList_MouseMove()`. The `processMouseMove()` method looks like this:

```
private void processMouseMove(System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDown)
    {
        if (e.Y >= ScrollingImageRectangle.Size.Height)
        {
            int delta = e.Y - ScrollingImageRectangle.Size.Height;
            delta = (delta / lineSpacing + 1) * lineSpacing;
            Point oldOrigin = originOfImage;
            originOfImage = new Point(
                originOfImage.X, Math.Min(
                    bitmap.Height - ScrollingImageRectangle.Size.Height,
                    oldOrigin.Y + delta));
            slide(oldOrigin, originOfImage);
        }
        if (e.Y < 0)
```

```

    int delta = -e.Y;
    delta = (delta / lineSpacing + 1) * lineSpacing;
    Point oldOrigin = originOfImage;
    originOfImage = new Point(
        originOfImage.X, Math.Max(0, oldOrigin.Y - delta));
    slide(oldOrigin, originOfImage);
}
adjustScrollBars();

int row = getRowFromHitTest(e);
if (row < draggingStartedRow)
{
    this.startingSelectedRow = row;
    this.endingSelectedRow = draggingStartedRow;
}
else
{
    this.startingSelectedRow = draggingStartedRow;
    this.endingSelectedRow = row;
}

if (old_startingSelectedRow != startingSelectedRow || old_endingSelectedRow != endingSelectedRow)
{
    drawIntoImage();
    this.Invalidate();
}
old_startingSelectedRow = startingSelectedRow;
old_endingSelectedRow = endingSelectedRow;
}
}

```

The first part of this method is the key part of this example. If the mouse has moved to above or below the client area of the control, we discover this by examining the `e.Y` value. In this case, we slide the scroll bars by an appropriate amount using a `slide()` method that we'll examine soon.

Then the bulk of the code for the `MouseMove` event handler is devoted to modifying the recorded set of selected rows, as indicated by the `startingSelectedRow` and `endingSelectedRow` fields, and repainting to ensure all those rows are drawn selected. Notice how the `old_startingSelectedRow` and `old_endingSelectedRow` fields are also used to temporarily store the range of selected rows for comparison the next time that a `MouseMove` event occurs, so we can look for any changes as a result of moving the mouse.

Now for the final part of the selection process—the point where the user releases the mouse button. Here, we simply set some fields to indicate that the drag-select operation is over:

```

private void ScrollAndSelectList_MouseUp(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    mouseDown = false;
    old_startingSelectedRow = -1;
    old_endingSelectedRow = -1;
}

```

Before we go on to examine the `MouseUp` event handler, let's look at that `slide()` method, which is designed to slowly scroll the control. Since the emphasis is on scrolling slowly, we use the `Thread.Sleep()` method to wait for ten milliseconds at the end of the process to slow down the scrolling:

```

private void slide(Point oldP, Point newP)
{
    Graphics g = this.CreateGraphics();

    if (oldP.Y != newP.Y)
    {

```

```

// Slide vertically.
bool goingUp = newP.Y - oldP.Y > 0;
int delta = (newP.Y - oldP.Y) / 12;
if (goingUp && delta < 1)
    delta = 1;
if (!goingUp && delta > -1)
    delta = -1;
int i = oldP.Y;
while (true)
{
    if (i == newP.Y)
        break;
    i += delta;
    if (goingUp && i > newP.Y)
    {
        i = newP.Y;
        continue;
    }
    if (!goingUp && i < newP.Y)
    {
        i = newP.Y;
        continue;
    }
    drawImage(g, oldP.X, i);
    if (i != newP.Y)
        Thread.Sleep(10);
}
}

```

Finally, we also need an event handler for when the user directly scrolls using the vertical scroll bar. This method simply examines the new position of the scroll bar and adjusts the part of the list box that is being viewed accordingly:

```
private void vScrollBar_Scroll( object sender, System.Windows.Forms.ScrollEventArgs e )
{
    switch (e.Type)
    {
        case ScrollEventType.SmallIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeIncrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.SmallDecrement:
            goto case ScrollEventType.LargeDecrement;
        case ScrollEventType.LargeDecrement:
            Point oldOrigin = originOfImage;
            originOfImage = new Point(originOfImage.X, e.NewValue);
            slide(oldOrigin, originOfImage);
            break;
        default:
            originOfImage = new Point(originOfImage.X, e.NewValue);
            this.Invalidate();
            break;
    }
}
```

That completes the control. We now need a test form to host it in and test it, so add a Windows Application to the current project and call it `TestScrollAndSelectList`. Set it as the startup project, add a reference in the test application to the custom control, and on its form, place a `ScrollAndSelectList` custom control. Change the font for the control to a larger setting, say 14-point Times New Roman. Also, the application will look better if you set the `Dock` property to `Fill`.

Run the program, select a row, and drag the mouse below the control. You will notice that the control scrolls as long as you wiggle the mouse. Now, let's modify the custom control and eliminate this bug.

Removing the Wiggle-the-Mouse Bug

To remove the bug, add the following private member variables to the `ScrollAndSelectList` custom control:

```
private bool autoDrag = false;
private Point lastMousePoint;
private System.Windows.Forms.Timer timer;
```

As you can see, we have a timer to measure how long the user holds the mouse outside the client area, a bool to indicate if we are in `autoDrag` mode (in other words, the user is selecting by holding the mouse outside the client area), and a `Point` field to store the last position of the mouse in this situation.

Next, modify the `MouseMove` event handler so that it can detect if the user has moved the mouse outside the client area; in which case, `autoDrag` needs to commence (or be stopped if the user has moved the mouse back into the client area):

```
private void ScrollAndSelectList_MouseMove(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    if (mouseDown)
    {
        if (e.Y >= ScrollingImageRectangle.Size.Height || e.Y < 0)
        {
            lastMousePoint = new Point(e.X, e.Y);
            if (!autoDrag)
                startAutoDrag();
        }
        else
        {
            if (autoDrag)
                stopAutoDrag();
            processMouseMove(e);
        }
    }
    else
        processMouseMove(e);
}
```

Note that the `MouseMove` event handler passes its event on to the `processMouseMove` method only if the control is not auto-dragging.

Here are the `startAutoDrag()` and `stopAutoDrag()` methods that deal with starting and stopping the timer:

```
private void startAutoDrag()
{
    autoDrag = true;
    timer = new System.Windows.Forms.Timer();
    timer.Interval = 100;
    timer.Tick += new EventHandler(tickEventHandler);
    timer.Start();
}

private void stopAutoDrag()

{
    autoDrag = false;
    if (timer != null)
    {
        timer.Stop();
        timer.Dispose();
        timer = null;
    }
}
```

Here is the handler for the `Tick` event in the timer, which simulates the mouse being moved by calling the `processMouseMove()` method to add some more rows to the selection:

```
private void tickEventHandler(Object o,
    EventArgs e)
{
    MouseEventArgs mea = new MouseEventArgs(MouseButtons.Left,
        1, lastMousePoint.X, lastMousePoint.Y, 0);
    processMouseMove(mea);
}
```

We must also modify the `MouseUp` event handler, adding a call to the `stopAutoDrag()` method:

```
private void ScrollAndSelectList_MouseUp(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    mouseDown = false;
    old_startingSelectedRow = -1;
    old_endingSelectedRow = -1;
    stopAutoDrag();
}
```

Now, recompile and rebuild the application. The application no longer exhibits the wiggle-the-mouse bug.

 [Previous](#)

[Next](#) 

 [Previous](#)

[Next](#) 

Summary

In this chapter, we examined some of the more advanced problems associated with processing mouse events. We looked at using mouse cursors to give a greater level of feedback to the end user. You can use the stock cursors, or you can create custom cursors.

You saw that hit testing is made easy by functionality in the `Rectangle` structure, and the `Region` and `GraphicsPath` classes.

You also saw the difference between drawing during a `Paint` event and drawing during a mouse event. You can get a `Graphics` object for the form during the mouse event and draw without waiting for a `Paint` event.

The chapter next covered how to draw directly to the display using an alpha technique, creating a semitransparent image of the object being dragged. We also examined the preventative measures to take so that you don't have the situation where your control thinks that the mouse button is down when it is not.

Finally, you saw how to code an application that allows the user to select parts of the display area by holding the mouse just outside the client area in a manner similar to how selection works in many text editors.

 [Previous](#)

[Next](#) 

 PreviousNext 

Appendix A: Using Namespaces

Overview

The classes that make up the GDI+ feature set are contained in six *namespaces* provided by the .NET Framework. When you're writing controls or testing examples that use the classes in these namespaces (or, indeed, in any of the .NET Framework's other namespaces), you need to state explicitly in your control's code that you intend to use those namespaces. If you don't, you'll get an error message like this when you try to compile the project:

The type or namespace name 'Drawing' does not exist in the class or namespace 'System' (are you missing an assembly reference?)

Let's take a look at a quick example to demonstrate the problem and the solution. Create a new Windows Application called TestNamespaces. Add a Paint event handler to the form, and then modify it like this:

```
private void Form1_Paint(object sender,
                         System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    GraphicsPath gp = new GraphicsPath();

    gp.AddLine(10, 10, 50, 10);
    gp.AddLine(50, 10, 50, 50);

    g.DrawPath(Pens.Black, gp);
}
```

Now attempt to compile the application. The attempted compilation should fail, and you should see four errors in the Task List window, as shown in [Figure A-1](#).

Error List				
Description	File	Line	Column	Project
① The type or namespace name 'GraphicsPath' could not be found (are you missing a using directive or an assembly reference?)	Form1.cs	21	2	TestNamespaces
② The type or namespace name 'GraphicsPath' could not be found (are you missing a using directive or an assembly reference?)	Form1.cs	21	29	TestNamespaces
③ The best overloaded method match for 'System.Drawing.Graphics.DrawPath(System.Drawing.Pen, System.Drawing.Drawing2D.GraphicsPath)' has some invalid arguments	Form1.cs	26	7	TestNamespaces
④ Argument '2' cannot convert from "GraphicsPath" to "System.Drawing.Drawing2D.GraphicsPath"	Form1.cs	26	30	TestNamespaces

Figure A-1: Error messages related to missing namespace information

In fact, these four errors all point to the same problem: you're trying to use an instance of the `GraphicsPath` class, which is part of a .NET Framework namespace called `System.Drawing.Drawing2D`. However, the application is not able to use anything that belongs to that namespace, because you did not explicitly state that it should use the contents of that namespace.

You can find the code that lists the included namespaces near the top of the code for the control:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

Each of these lines is a `using` directive, and each line specifies a different namespace that you would like the application to use. As you can see, seven namespaces are listed, but `System.Drawing.Drawing2D` is not among them. So, to solve this problem, you simply need to add another `using` directive, like this:

```
using System.Drawing.Drawing2D;
```

After you add this `using` statement, try compiling the code again. The code should compile this time. When it runs, you should see a simple image with a couple of lines, as shown in [Figure A-2](#).

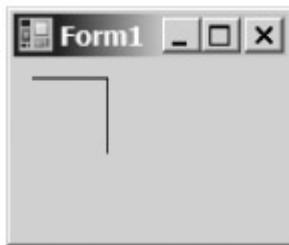


Figure A-2: Result of drawing a `GraphicsPath` object

Note Over the course of the book, the examples use the six namespaces `System.Drawing`, `System.Drawing.Drawing2D`, `System.Drawing.Imaging`, `System.Drawing.Printing`, `System.Drawing.Text`, and `System.Drawing.Design`. The instructions indicate the namespaces involved, so that you can insert the correct `using` directives in your application. You'll find information about the contents of each namespace in the MSDN online documentation. Try http://msdn.microsoft.com/library/en-us/cpref/html/cpref_start.asp as a starting point.

It's reasonable to ask where the `System.Drawing.Drawing2D` namespace comes from. Are all namespaces available in the same way? The answer is yes, if you make them available.

As noted earlier, the `System.Drawing.Drawing2D` namespace is contained in an assembly called `System.Drawing.dll`. The namespace is available to you because the application contains a reference to the assembly. To see this, take a look at the Solution Explorer in your IDE window. Under the References node, you'll see a number of references, one of which is `System.Drawing`. If you look at the properties of this reference (by highlighting `System.Drawing` and pressing F4 to bring up the Properties window), you'll see that this is a reference to the file `System.Drawing.dll`, as shown in [Figure A-3](#). It's this reference that makes the `System.Drawing.Drawing2D` namespace available to you through a simple `using` directive in your code.



Figure A-3: Properties of the `System.Drawing` namespace

In fact, the Visual Studio .NET IDE includes a reference to the `System.Drawing.dll` library file by

default whenever it creates a Windows Application (because it expects that the application is going to involve some amount of drawing activity). If you create a Console Application instead, you will find that it does not include a reference to `System.Drawing.dll`.

Adding a reference to a type library is easy. To do so, in the Solution Explorer, right-click the References node and select Add Reference. This will bring up the Add Reference dialog box, as shown in [Figure A-4](#). In this dialog box, navigate to the DLL or assembly that you need and select it.

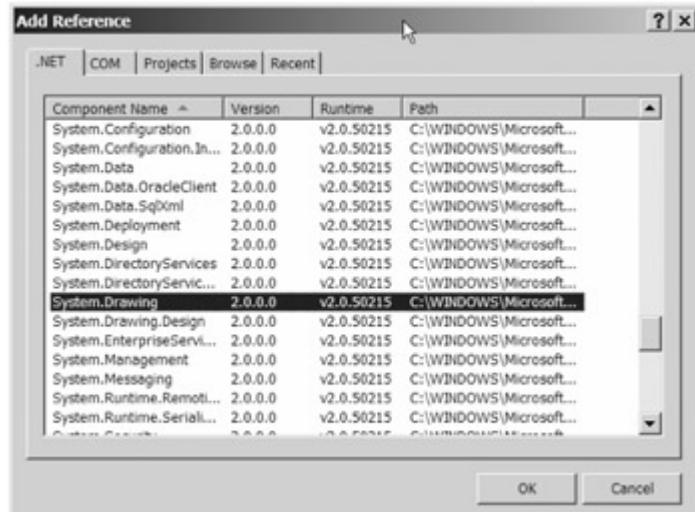


Figure A-4: Add Reference dialog box

Once you have added the desired reference to your project, you can use any of the namespaces from that DLL.

[Previous](#)

[Next](#)

[Previous](#)[Next](#)

Appendix B: Using the Console in Windows Applications

Overview

You may find it handy to be able to write output to the console window, even in a Windows application. For example, using the console is particularly useful when you're debugging an application.

You could output information to message boxes or to some other GUI, but sometimes that really isn't an option. Using message boxes for this type of information is particularly cumbersome and intrusive. Also, when you're tracing a bug that is related to the order of events, using another message box or GUI will affect the order of events. Indeed, the debugger itself can alter the order of events. In these cases, sending output to the console window is a very useful alternative, and one that I use for examples in this book.

To write text output to the console window, use the `Console` class. The `Console` class is part of the `System` namespace, so the first thing you need to do is check that that the `System.dll` library is referenced in your application, as shown in [Figure B-1](#).

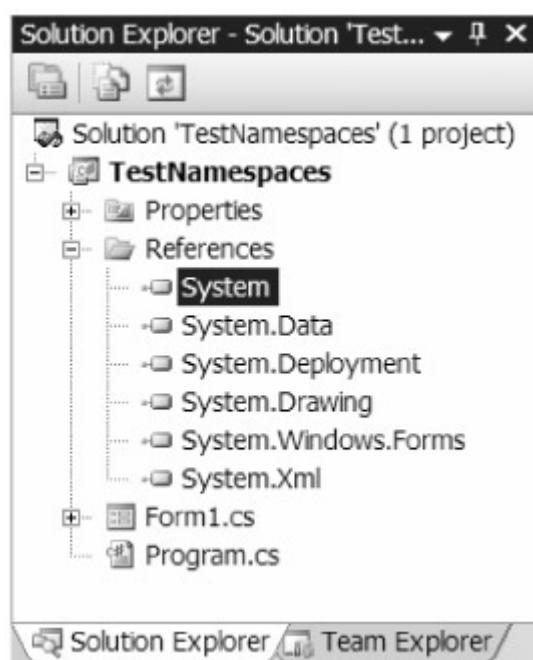


Figure B-1: Verifying that the `System.dll` library is referenced

Then make sure that your code includes a `using` directive for this namespace:

```
using System;
```

With the `System` reference and `using` directive in place, using the `Console` class in a Windows application is easy.

Note The `Debug` and `Trace` classes that come with the .NET Framework have a lot more functionality than the `Console` class. However, to keep this aspect of the code as simple as possible, I show you how to use the `Console` class to write text output to the console window.

To see how this works, create a new Windows application. Add a `Load` event handler to the form, and modify it so that it writes some text to the console, as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
```

```
Console.WriteLine("This text is written to the console " +
    "when the form is loaded...");
```

}

To see the console window, you need to change the output type of the project to Console Application. You do this through the Project Properties dialog box. Right-click the project node in the Solution Explorer and select Properties from the context menu, as shown in [Figure B-2](#).

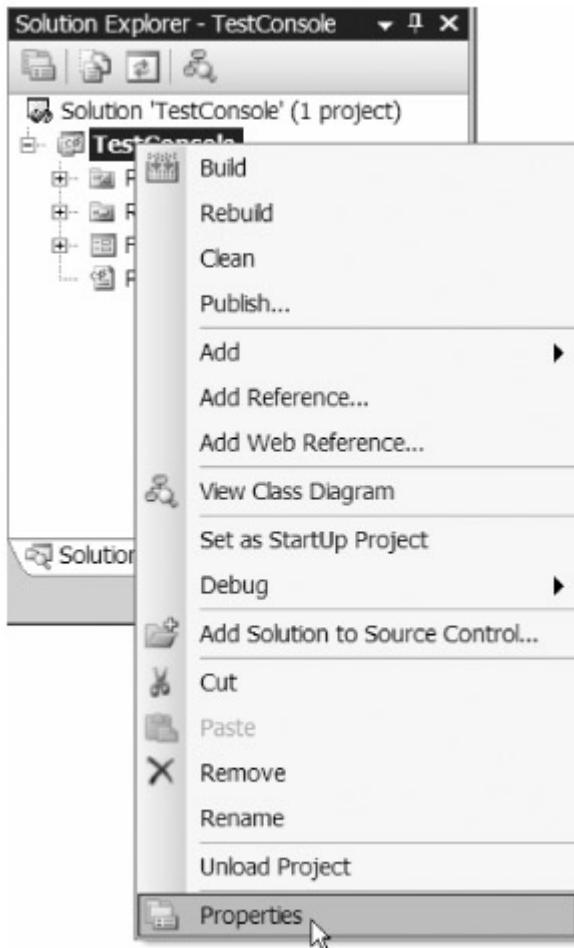


Figure B-2: Setting the properties of a project

In the dialog box that appears, change the Output type option to Console Application, as shown in [Figure B-3](#). This will allow you to print to the console window, but not prevent you from creating windows and dialog boxes.

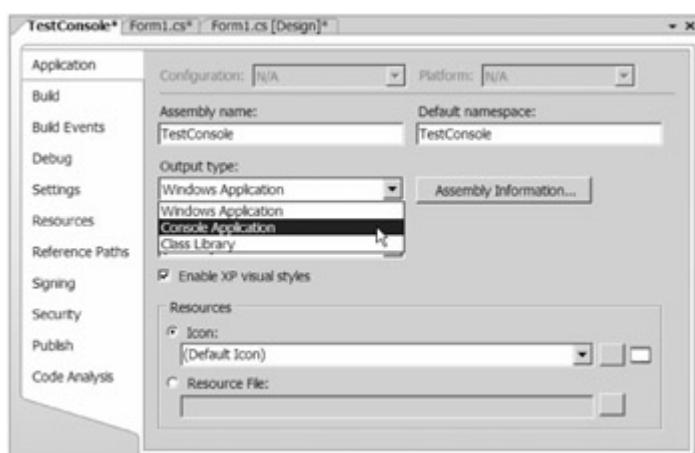


Figure B-3: Changing the output type to Console Application

When you run the application, you'll see a command prompt window (in addition to the graphical windows and controls). Everything that you write using the `Console` class appears in that window, as shown in [Figure B-4](#).

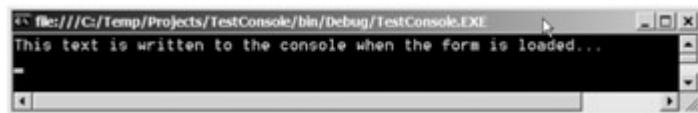


Figure B-4: Output to the console

That is all there is to it. As you can see, it's a simple technique.

[Previous](#)

[Next](#)

 PreviousNext 

Appendix C: Using the Dispose Method with Classes that Implement IDisposable

Overview

Some classes in the .NET Framework use Windows resources. In the context of Windows programming, the term *resource* refers to a special kind of memory used for certain types of operations. In the early days of Windows programming, this special memory was very limited. As soon as your computer ran out of resources, strange things would start to happen—the Windows operating system would begin to run very slowly, and windows would not be drawn correctly.

In those days, it was incumbent upon any application to free resources as soon as possible after it had finished using them. With the introduction of the Windows 9x operating systems (including Windows Millennium Edition), Microsoft developers increased the amount of resource memory available to applications, but they did not make it unlimited. Windows 2000 (according to its documentation) does not have any finite limit on resources; however, I have seen Windows 2000 misbehave when too many applications are open, and I've sometimes been able to restore correct behavior by closing some applications.

In the .NET Framework, many classes implement the `IDisposable` interface, which includes the `Dispose()` method. When you're using an instance of one of these classes, it is highly desirable to call its `Dispose()` method as soon as you are finished with the class, because the object releases its Windows resources at this point.

The `Dispose()` method is called in the destructor of any such object, but that does not remove the need to call `Dispose()` explicitly. The reason for this is related to the way that destruction occurs in managed languages. In unmanaged C++, when the object is deleted, the destructor is called immediately. However, in C# and other managed languages, it's the responsibility of the *garbage collector* to call the destructor, and you can't be certain when the garbage collector will run. In particular, if you have machines with a large amount of memory (which many do these days), the garbage collector may run *very* infrequently. This is not a big problem on Windows 2000 and XP systems, but if your application is running on Windows 9x, it very well may run out of resources before the computer runs out of memory. Running out of *memory* triggers the garbage collector to run, but running out of *resources* does not.

The .NET Framework runs on the Windows 9x operating systems, so if you are planning on distributing your application on that series of operating systems, it is important to call `Dispose()` whenever you are finished with an object that uses the `IDisposable` interface. Of course, it may be less important if you are not targeting those operating systems, but it is still good programming practice to release your resources when you are finished with them, even on Windows 2000 and XP.

Suppose you created the following GDI+ code, contained in a `Paint` event handler:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Aquamarine, 5);
    g.DrawLine(p, 10, 10, 50, 50);
    p.Dispose();
}
```

This code uses a `Graphics` object and a `Pen` object. The issue here is that both the `Graphics` class and the `Pen` class implement the `IDisposable` interface (you can confirm this by checking the documentation), but this code calls the `Dispose()` method *only* on the `Pen` object.

In fact, this is intentional and correct. The reason is that the `Graphics` object was created by the .NET Framework *before* it sent the `Paint` event; it is a property of the `PaintEventArgs` object that is passed into the event handler. Therefore, you did not create the `Graphics` object itself within your

event handler, so you shouldn't dispose of it. In contrast, the `Pen` object is your own creation, and so it's correct for you to dispose of it when you're finished with it.

Note The rule is that if your method *created* it, then your method should *dispose* of it.

If you're applying a C# `using` block within your code, and you create objects within the `using` block, the situation is slightly different. You're still responsible for disposing of objects that you create, but you don't need to call the `Dispose()` method explicitly, because C# does it at the end of the `using` block (when the object goes out of scope). The following code demonstrates this:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    using (Pen p = new Pen(Color.Aquamarine, 5))
    {
        g.DrawLine(p, 10, 10, 50, 50);
    }
}
```

According to the .NET documentation, the preceding code is precisely equivalent to the following:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Aquamarine, 5);
    try {
        g.DrawLine(p, 10, 10, 50, 50);
    }
    finally {
        if (p != null)
            ((IDisposable)p).Dispose();
    }
}
```

You can create multiple objects in the `using` block, separating them by commas; however, they need to be of the same type:

```
using (Pen p = new Pen(Color.Aquamarine, 5),
      p2 = new Pen(Color.Red))
```

Note If you're unfamiliar with the C# language structure, you should note the two distinct uses for the `using` keyword in the language. The first is in the form of a `using` block (which delimits scope, as shown here). The second is the `using` directive (which allows you to use the types of a namespace without spelling out the full namespace name of the type each time you use it).

The only possible complaint that you might have with the `using` block is that it creates another level of indentation, but the benefits are threefold:

- You know that the object can't be disposed of before its time.
- You can be sure that the resource is released, even if an exception is raised.
- The `using` block avoids the possibility that the resource has been disposed of twice, by making sure that the reference is not `null`.

As you go through the book, you will notice that many of the examples dispose of objects or use the `using` construct.