QUE® CERTIFICATION

Windows Programming (Level 1) (C#1)

Developing and Implementing Windows-Based Applications with Microsoft® Visual C#™ .NET and Microsoft® Visual Studio® .NET

Contents

Table of Contents

OBJECTIVES

This chapter covers the following Microsoft-specified objectives for the "Creating User Services" section of Exam 70-316, "Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET":

Create a Windows form by using the Windows Forms Designer:

- **Add and set properties on a Windows form.**
- **Create a Windows form by using visual inheritance.**
- **Build graphical interface elements by using the `System.Drawing` namespace.**

▶ Windows forms are the basic user interface element of a Windows application. The exam objectives addressed in this chapter cover the basics of designing a Windows form by using the Windows Forms Designer. This exam objective addresses the following specific topics:

- How to create a Windows form and change its behavior and appearance through its built-in properties and through custom-added properties.
- How to use visual inheritance to rapidly design a Windows form by inheriting it from an existing Windows form.
- How to build various graphical interface elements by using the System.Drawing namespace.

Create, implement, and handle events.

▶ Event handling is the core part of programming a user interface. This chapter describes how to make a Windows form respond to user actions. You'll find further coverage of this exam objective in Chapter 4, "Creating and Managing .NET Components and Assemblies."

CHAPTER 1

Introducing Windows Forms

OUTLINE

Study Strategies

- Make yourself comfortable with the major properties of Windows forms. This chapter's examples and exercises introduce the most important form properties.
- Invest time looking at and understanding the code that is automatically generated by Visual Studio .NET for you.
- Make sure you fully understand event handling. This will enable you to write interactive Windows applications.
- Experiment with classes in the `System.Drawing` namespace. In addition to the completing the examples and exercises in this chapter, it would be a good idea for you to create a small sample program to test the behavior of a class or a property whenever you are in doubt.
- If you are new to object-oriented programming, consider reading all or some of the recommended material listed in the "Suggested Readings and Resources" section at the end of this chapter.

04 0789728230 CH01 11/21/02 1:24 PM Page 24

INTRODUCTION

In this chapter, the first step toward passing Exam 70-316, you will complete a lot of the groundwork required to build the foundation for the rest of this book.

This chapter starts with an overview of the .NET Framework and various development tools for developing applications for the .NET Framework. This overview will be enough to get you started; I'll continually cover advanced features as they become important for meeting exam objectives.

Next, this chapter talks about designing Windows forms, both by using a visual designer and by manually writing code. The visual designer that is built inside Visual Studio .NET helps you rapidly develop forms. As you design forms, you will also learn about many useful classes that are available in the `System.Windows.Forms` namespace. You will also learn how to visually inherit a Windows form from an existing form.

A user can generally interact with a Windows application. Applications can respond to users' actions thanks to event handling. In this chapter you will learn how to make programs interactive by using event handling.

Finally, this chapter talks about the various classes in the `System.Drawing` namespace. These classes allow you to add typography, 2-D graphics, and imaging features to applications.

KEY CONCEPTS

In this book, you will develop Windows application using Visual Studio .NET. Under the hood, you will be using the Framework Class Libraries (FCL) to write applications that run on the Common Language Runtime (CLR). Both FCL and CLR are part of a larger framework called the .NET Framework. In this section, I'll give you an overview of the .NET Framework, various development tools, and basic object-oriented concepts that you will need right from the beginning.

An Overview of the .NET Framework

The Microsoft .NET Framework is a new computing platform for developing distributed applications. It provides several new features that enhance application development. The following are some of these features:

- **Consistent development model**—The .NET Framework proves an object-oriented and consistent development model. When you learn programming in the .NET Framework, you can use your skills in developing different types of applications, such as Windows Forms applications, Web applications, and Web services.
- **Robust execution environment**—The .NET Framework provides an execution environment that maximizes security, robustness, and performance of applications while minimizing deployment and versioning conflicts.
- **Support for standards**—The .NET Framework is built around industry standards such as Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Common Language Infrastructure (CLI), and C#.

Because the .NET Framework provides new execution environments for running the applications designed for the .NET Framework, you need to install the .NET Framework on the target machine. The .NET Framework can be installed using the .NET Framework redistributable file (approximately 21MB). You can find the link to download this file from the Microsoft official Windows Forms Community site (`www.windowsforms.net`).

The .NET Framework has two main components:

- The Common Language Runtime (CLR)
- The Framework Class Library (FCL)

The Common Language Runtime

The CLR provides a managed and language agnostic environment for executing applications designed for the .NET Framework.

The managed runtime environment provides several services to the executing code: compilation, code safety verification, code execution, automatic memory management, and other system services. The applications designed to run under the CLR are known as managed applications because they enjoy the benefit of services offered by the managed execution environment provided by the CLR.

The CLR is based on the Common Language Infrastructure (CLI). CLI provides a rich type system that supports the types and operations found in many programming languages. If a language compiler adheres to the CLI specifications, it can generate code that can run and interoperate with other code that executes on the CLR. This allows programmers to write applications using a development language of their choice and at the same time take full advantage of the CLR, FCL, and components written by other developers.

NOTE

C# and CLI Are ECMA Standards
The C# programming language, as well as the CLI, are ECMA standards. The standardization has motivated several vendors to support and extend the .NET Framework in various ways. Some example includes the Mono project (`www.go-mono.com`), which is an open-source implementation of the .NET Framework; the Delphi 7 Studio (`www.borland.com/delphi`), which brings Delphi language to the .NET Framework; and Covalent Enterprise Ready Servers (`www.covalent.net`), which supports ASP.NET on the Apache Web server.

Microsoft provides five language compilers for the .NET Framework: Visual C# .NET, Visual Basic .NET, Managed C++ .NET, Jscript .NET, and J# .NET. When you install the .NET Framework, you get only command-line compilers for C#, Visual Basic .NET, and Jscript .NET. The Managed C++ compiler is part of .NET Framework SDK and Visual Studio .NET, whereas the J# .NET compiler can be downloaded separately from the Microsoft Web site. Visual J# .NET will ship as a component of Visual Studio .NET starting with version 1.1. In addition to the compilers available from Microsoft, you can obtain CLI-compliant compilers for languages such as COBOL, Delphi, Eiffel, Perl, Python, Smalltalk, and Haskell from various independent vendors or organizations.

The CLI-compliant language compilers compile the source language to an intermediate format known as the Common Intermediate Language (CIL). At runtime, the CLR compiles the CIL code to the machine-specific native code using a technique called *just-in-time (JIT)* compilation. Microsoft's implementation of CIL is the *Microsoft Intermediate Language (MSIL)*.

The Framework Class Library

The FCL is an extensive collection of reusable types that allows you to develop a variety of applications, including

- Console applications
- Scripted or hosted applications

- Desktop applications (Windows Forms)
- Web applications (ASP.NET applications)
- XML Web services
- Windows services

The FCL organizes its classes in hierarchical namespaces so that they are logically grouped and easy to identify. You will learn about several of these namespaces and the classes that relate to the Windows Forms applications in this book.

An Overview of the Development Tools

Two development tools are available from Microsoft to help you design the Windows applications that run on the .NET Framework:

- The .NET Framework SDK
- Visual Studio .NET

I discuss these tools in the following sections.

The .NET Framework SDK

The Microsoft .NET Framework Software Development Kit (SDK) is available as a free download (about 131MB). You can find the link to download it from `www.windowsforms.net`. When you install the .NET Framework SDK you get a rich set of resources to help you develop applications for the Microsoft .NET Framework. This includes

- **The .NET Framework**—This installs the necessary infrastructure of the .NET Framework, including the CLR and the FCL.
- **Language compilers**—The command-line–based compilers allow you to compile your applications. The language compilers installed with the SDK are Visual C# .NET, Visual Basic .NET, Jscript .NET, and a non-optimizing compiler for Managed C++ .NET.
- **Tools and debuggers**—Various tools installed with the .NET Framework SDK make it easy to create, debug, profile, deploy, configure, and manage applications and components. I discuss most of these tools as I progress through this book.

- **Documentation**—The .NET Framework SDK installs a rich set of documentation to quickly get you up to speed on development using the .NET Framework. These include the QuickStart tutorials, product documentation, and samples.

It's possible to develop all your programs by using just a text editor and the command-line compilers and tools provided by the .NET Framework SDK. However, Visual Studio .NET provides a much more productive development environment.

Visual Studio .NET

Visual Studio .NET provides developers with a full-service Integrated Development Environment (IDE) for building Windows Forms applications, ASP.NET Web applications, XML Web services, and mobile applications for the .NET Framework. Visual Studio .NET supports multiple languages—Visual C# .NET, Visual Basic .NET, Visual C++ .NET, and Visual J# .NET—and provides transparent development and debugging facilities across these languages in a multilanguage solution. Additional languages from other vendors can also be installed seamlessly into the Visual Studio .NET shell.

Visual Studio .NET installs the .NET Framework SDK as a part of its installation. In addition to the SDK features, some important features of Visual Studio .NET are

- **IDE**—Supports development, compilation, debugging, and deployment, all from within the development environment.
- **Editing tools**—Supports language syntaxes for multiple languages. The IntelliSense feature provides help with syntax. Visual Studio .NET also supports editing of XML, Extensible Stylesheet Language (XSL), Hypertext Markup Language (HTML), and Cascading Style Sheets (CSS) documents, among other types.
- **Integrated debugging**—Supports cross-language debugging, including debugging of SQL Server stored procedures. It can seamlessly debug applications that are running locally or on a remote server.

- **Deployment tools**—Support Windows Installer. These tools also provide graphical deployment editors that allow you to visually control various deployment settings for Visual Studio .NET projects.
- **Automation**—Provides tools for extending, customizing, and automating the Visual Studio .NET IDE.

You will learn about all of these features in the course of this book. As an exam requirement, this book uses Visual Studio .NET as its preferred tool for developing Windows Forms applications.

EXAM TIP

Using the IDE Exam 70-316 requires you to know Visual Studio .NET and Visual C# .NET programming language for Windows-based application development. You might be asked questions about specific Visual Studio .NET features.

Understanding Classes, Inheritance, and Namespaces

The .NET Framework is designed to be object oriented from the ground up. I'll cover the different elements of object-oriented programming as they come up, but before I start, you should know a few terms, such as class, inheritance, and namespace that are important right from the beginning. The following sections briefly explain these terms and their meaning.

Classes

C# is an object-oriented programming language. One of the tasks of a C# developer is to create user-defined types called *classes*. A class is a reference type that encapsulates data (such as constants and fields) and defines its behaviors using programming contructs such as methods, properties, constructors, and events.

A class represents an abstract idea that you would like to include in an application. For example, the .NET Framework includes a `Form` class, which includes data fields for storing information such as the size of the form, the form's location, the form's background color, title bar text, and so on. The `Form` class also contains methods that define how a form behaves, such as a `Show()` method that shows the form onscreen and an `Activate()` method that activates the form by giving it the focus.

A class functions as the blueprint of a concept. When you want to work with a class in a program, you create instances of the class, which are called *objects*. Objects are created from the blueprint defined by the class, but they physically exist in the sense that they have memory locations allocated to them and they respond to messages. For example, to create an actual form in a program, you create an instance of the `Form` class. After you have that instance available, you can actually work on it—you can set its properties and call methods on it.

Each object maintains its own copy of the data that is defined by the class. This allows different instances of a class to have different data values. For example, if you have two instances of the class `Human`—`objYou` and `objMe`—these two objects can each have a different value for their `EyeColor` properties. You access the member of an object by using *`ObjectName`*`.`*`MemberName`* syntax, where *`ObjectName`* is name of the class instance and *`MemberName`* can be a field, a property, a method, or an event. When an object is created, it creates its members in a special area in memory called the *heap,* and it stores a pointer to that memory. Because classes use pointers to refer to their data, they are sometimes also called *reference types*.

NOTE

Static Members of a Class A class can have static members (fields, methods, and so on). Static members belong to the class itself rather than to a particular instance. No instance of a class is required in order to access its static members. When you access a static member of a class, you do so by prefixing its name with the name of the class—for example, `ClassName.StaticMemberName`.

In contrast with the reference types, C# also has a structure type (called a *struct*), which is defined by using the `struct` keyword. Structs are similar to classes, but rather than store a pointer to the memory location, a struct uses the memory location to store its members. A struct is also referred to as a *value type*.

Among the members of classes, properties warrant special attention. A property provides access to the characteristics of a class or an instance of that class (that is, an object). Examples of properties include the caption of a window, the name of an item, and the font of a string.

To the programs using a class, a property looks like a field—that is, a storage location. Properties and fields have the same usage syntax, but their implementations differ. In a class, a property is not a storage location; rather, it defines accessors that contain code to be executed when the property value is being read or written. This piece of code allows properties to preprocess the data before it is read or written, to ensure integrity of a class. Using properties is the preferred way of exposing attributes or characteristics of a class, and various classes in this chapter use properties extensively.

04 0789728230 CH01 11/21/02 1:24 PM Page 31

Inheritance

Object-oriented programming languages such as C# provide a feature called inheritance. *Inheritance* allows you to create new types that are based on types that already exist. The original type is called a *base class,* and the inherited class is called a *derived class*. When one class inherits from another class, the derived class gets all the functionality of the base class. The derived class can also choose to extend the base class by introducing new data and behavioral elements. In developing Windows forms, you will frequently inherit from the `Form` class to create your own custom forms; these custom forms will be at least as functional as an object of the `Form` class, even if you do not write any new code in the derived class. Value types such as structs cannot be used for inheritance.

It is interesting to note that every single type (other than the `Object` class itself) that you create or that is already defined in the framework is implicitly derived from the `Object` class of the `System` namespace. This is the case to ensure that all classes provide a common minimum functionality. Also note that a C# type can inherit from only a single parent class at a time.

Inheritance is widely used in the FCL, and you will come across classes (for example, the `Form` class) that get their functionality from other classes (for example, the `Control` class) as a result of a chain of inheritances.

NOTE

Access Modifiers A class can define the accessibility of its member by including an access modifier in its declaration. C# has four different access modifiers:

- **`public`**—Allows the member to be globally accessible.
- **`private`**—Limits the member's access to only the containing type.
- **`protected`**—Limits the member's access to the containing type and all classes derived from the containing type.
- **`internal`**—Limits the member's access to within the current project.

Namespaces

Several hundred classes are available in the FCL. In addition, an increasingly large number of classes are available through independent component vendors. Also, you can develop classes on your own. Having a large number of classes not only makes organization impossible but can also create naming conflicts between various vendors. The .NET Framework provides a feature called a *namespace* that allows you to organize classes hierarchically in logical groups based on what they do and where they originate. Not only does a namespace organize classes, but it also helps avoid naming conflicts between vendors because each classname is required to be unique only within its namespace. A general convention is to create a namespace like this:

```
CompanyName.ApplicationName
```

04 0789728230 CH01 11/21/02 1:24 PM Page 32

In this case *`CompanyName`* is your unique company name and *`ApplicationName`* is a unique application name within the company. All classes related to this application then belong to this namespace. A class is therefore identified, for example, as `QueCertifications.Exam70316.ExamQuestions`, where `QueCertifications` is the unique name for a company, `Exam70316` is a unique application within that company, and `ExamQuestions` is the name of a specific class. `QueCertifications` could have another class with the same name, `ExamQuestions`, as long as it belongs to a different application, such as `QueCertifications.Exam70306`. The objective of namespaces is to keep the complete naming hierarchy unique so that there are no naming conflicts.

A namespace is a string in which dots help create a hierarchy. In the namespace `QueCertifications.Exam70316`, `Exam70316` is called a *child namespace* of `QueCertifications`. You could organize classes at two levels here: at the level of `QueCertifications` and also at the level of `Exam70316`. You can create a hierarchy with as many levels as you want.

NOTE

Namespace Hierarchies Versus Inheritance Hierarchy A namespace hierarchy has nothing to do with inheritance hierarchy. When one class inherits from another, the base class and the derived class may belong to different and unrelated namespaces.

A System namespace in the FCL acts as the root namespace for all the fundamental and base classes defined inside the FCL. One of the fundamental classes defined in the `System` namespace is `Object` class (uniquely identified as `System.Object`). This class acts as the ultimate base class for all other types in the .NET Framework.

The `System.Windows.Forms` namespace organizes classes for working with Windows forms. The `System.Drawing` namespace organizes classes for creating graphical elements. You will make use of many classes from these two namespaces in this chapter.

CREATING A WINDOWS FORMS APPLICATION

In this section, you will learn how to use Visual Studio .NET to create a Windows Forms application. In the process, you will become familiar with the development environment and a few common classes you will use with Windows Forms applications.

Using the `System.Windows.Forms.Form` Class

A Windows application generally consists of one or more Windows forms. A Windows form is an area on the screen (usually rectangular) over which you design the user interface of a Windows application. This area acts as a placeholder for various user interface elements, such as text boxes, buttons, lists, grids, menus, and scrollbars.

Programmatically speaking, a Windows form is an instance of the `Form` class of the `System.Windows.Forms` namespace. The `Form` class derives from the inheritance hierarchy shown in Figure 1.1.

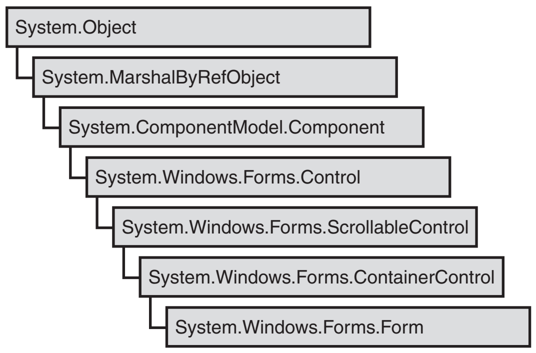

System.Object
System.MarshalByRefObject
System.ComponentModel.Component
System.Windows.Forms.Control
System.Windows.Forms.ScrollableControl
System.Windows.Forms.ContainerControl
System.Windows.Forms.Form

FIGURE 1.1
The `Form` class ultimately inherits from the Object class through an inheritance hierarchy.

The following points relate to the inheritance hierarchy shown in Figure 1.1 and the `Form` class:

- The `Form` class inherits from the `ContainerControl` class. By virtue of this inheritance, the `Form` class becomes capable of acting as a placeholder for various user interface elements or controls, such as text boxes, labels, buttons, and toolbars.
- A form is also a control, but it is a special type of control that is both scrollable and capable of acting as a container control. This is because the `Form` class inherits from `Control` class via the `ScrollableControl` and `ContainerControl` classes.
- The `Form` class is ultimately derived from the `Object` class of the `System` namespace, just like any other class. As a result of this inheritance, you can also say that a form is of type `Object`.

- As a result of inheritance, the `Form` class has access to several members (methods, properties, events, and so on) that are available to it through its parent classes. The `Form` class also adds a set of new members for its specific functionality. This is typical of the way that inheritance hierarchies work.

You can use the properties of a `Form` object (such as `Size`, `BackColor`, and `Opacity`) to modify the way a form appears onscreen. Methods of a `Form` object can be used to perform on the form actions such as Show and Hide. You can also attach to the form custom code that acts as event handlers; this enables the form to respond to different actions performed on the form.

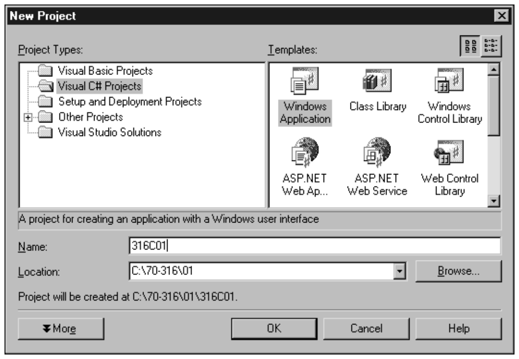
Designing a Windows Form by Using the Windows Forms Designer

Visual Studio .NET provides the Windows Forms Designer (sometimes also called *the designer* or *the visual designer*) for designing Windows forms. To get started with the designer, you can use it to create a simple Windows form. The simple exercise shown in Step by Step 1.1 helps you become familiar with different pieces of the development environment you use to create Windows forms.

STEP BY STEP

1.1 Creating a Windows Form

1. Launch Visual Studio .NET. On the start page, click the New Project button (alternatively, you can select File, New, Project). In the New Project dialog box, select Visual C# Project as the project type and Windows Application as the template. Name the project `316C01`, as shown in Figure 1.2.



A project for creating an application with a Windows user interface

Name: 316C01

Location: C:\70-316\01

Project will be created at C:\70-316\01\316C01.

FIGURE 1.2
You can create a new Visual C# Windows application by choosing the Windows Application template for your Visual C# project.

2. The development environment is now in the design view (see Figure 1.3), and you are shown an empty form. The Solution Explorer window (see Figure 1.4) allows you to see all files that Visual Studio .NET includes in the project. If the Solution Explorer is not already visible, you can invoke it by selecting View, Solution Explorer. `Form1.cs` is the file that stores the C# code for the default Windows form that's created as part of a new project. Right-click `Form1.cs` and select Rename from the context menu. Rename the file `StepByStep1_1.cs`.

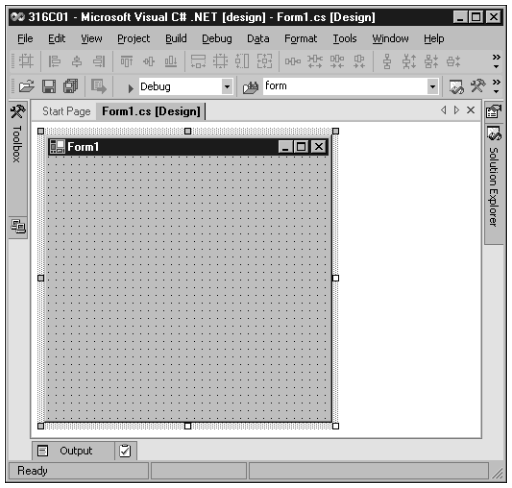


FIGURE 1.3◀
The Windows Forms Designer Environment enables you to visually develop a Windows application.

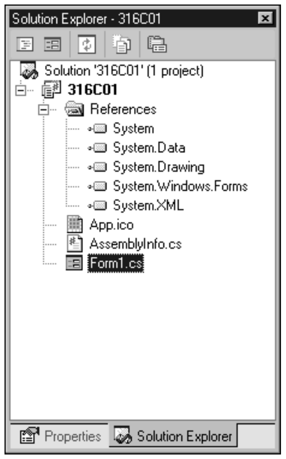
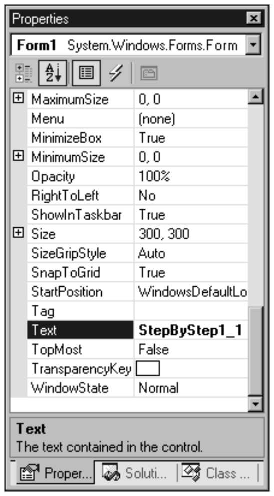


FIGURE 1.4▲
You can use the Solution Explorer to manage files within a Visual Studio .NET solution.



Form1 System.Windows.Forms.Form

⊞ MaximumSize	0, 0
Menu	(none)
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
RightToLeft	No
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
SnapToGrid	True
StartPosition	WindowsDefaultLo
Tag	
Text	StepByStep1_1
TopMost	False
TransparencyKey	☐
WindowState	Normal

Text
The text contained in the control.

FIGURE 1.5
The Properties window shows the properties of an object.

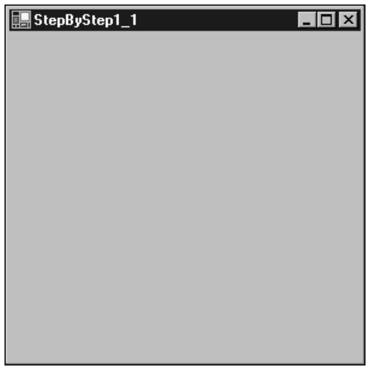


FIGURE 1.6
This simple Windows form shows customized title bar text.

3. The form's title bar displays the text `Form1`. `Title` is a property of a form, and you can manipulate `Title` through the Properties window. Click the form so that it gets the focus, and then press F4 or select View, Properties Window. Change the `Text` property of the form in the Properties window to `StepByStep1_1`, as shown in Figure 1.5. The form's title bar now displays `StepByStep1_1`.

4. Select Debug, Start or click F5 to execute the project; this displays your very first Windows form. You should see something similar to Figure 1.6. Try positioning this form anywhere onscreen by dragging its title bar. You can increase or decrease the form size by dragging the form's border.

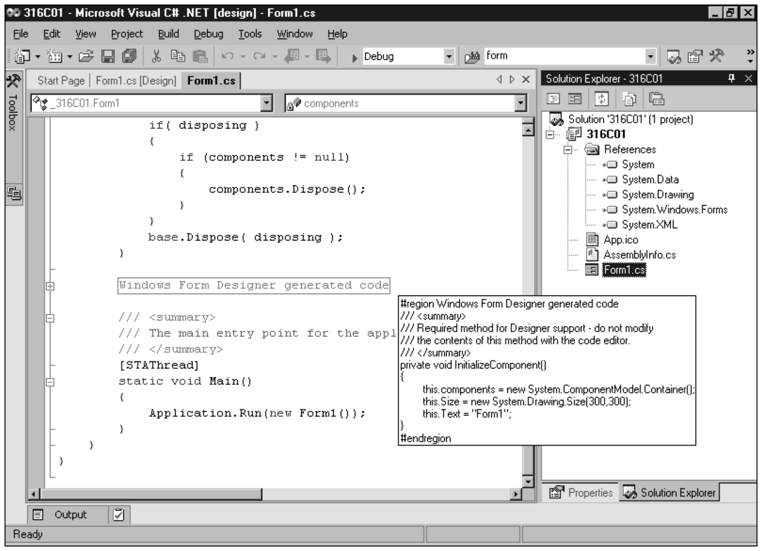
5. Click the close button of the form to end the execution of this application and return to the design view.

6. Right-click anywhere on the form and select View Code from the context menu. This opens a new window in the Visual Studio .NET environment, showing the code corresponding to the Windows form. Click the + sign next to the Windows Form Designer Generated Code region and observe all the code that the designer automatically generated for you.

In Step by Step 1.1, when you create a new Windows application, Visual Studio .NET automatically includes a Windows form inside it. The Windows form contains the code to launch itself when you run the project.

Each Windows form resides in a code file whose filename extension depends on the language you are using (for example, `.cs` for C#). Visual Studio .NET originally assigned the name `Form1.cs` to the code file, but you changed it to `StepByStep1_1.cs` through the Solution Explorer. This code file contains the definition of the Windows form that you saw when you executed the project.

In the Windows Forms Designer, a Windows form can be seen in two views: the design view (see Figure 1.3) and the code view (see Figure 1.7). What you see in the design view is nothing but a visual representation of the code. When you manipulate the form by using the designer, this code is automatically generated or modified based on your actions. You can also write the complete code yourself in the code view. When you switch back to the design view, the designer reads the code in order to draw the corresponding form onscreen.


316C01 - Microsoft Visual C# .NET [design] - Form1.cs
File Edit View Project Build Debug Tools Window Help
Debug
form
Start Page | Form1.cs [Design] | Form1.cs
_316C01.Form1
components
Toolbox
Solution Explorer - 316C01
Solution '316C01' (1 project)
316C01
References
System
System.Data
System.Drawing
System.Windows.Forms
System.XML
App.ico
AssemblyInfo.cs
Form1.cs
Properties
Solution Explorer
Output
Ready

```
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        Windows Form Designer generated code

        /// <summary>
        /// The main entry point for the appl
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
        this.components = new System.ComponentModel.Container();
        this.Size = new System.Drawing.Size(300,300);
        this.Text = "Form1";
}
#endregion
```

FIGURE 1.7
The code view allows you to view and modify the code associated with a Windows form.

NOTE

Projects and Solutions A *solution* is used to group one or more projects. In a typical application you first create a solution and then add projects to it. If you directly create a project, Visual Studio .NET automatically creates a solution for it. In that case, the name of the solution defaults to the name of project. For example, the project `316C01` is automatically created in solution `316C01`.

Exploring the Generated Code

While exploring the code in Step by Step 1.1 you probably noticed that Visual Studio .NET groups code into blocks. This feature is called *code outlining*. You can expand and collapse code blocks by using the + and - signs near the left margin of the window in code view (see Figure 1.7). Code outlining is especially helpful when you are working with large code files. You can collapse certain areas of code that you don't want to focus on at that time and continue editing the expanded sections in which you are interested.

Step by Step 1.1 also shows a rectangular block marked Windows Form Designer Generated Code. This is a block of collapsed code that has a name. When you expand the block, you see a set of statements included between `#region` and `#endregion` directives. These directives mark the beginning and end of a named code block. You can specify a name after the `#region` directive to identify the code block with a name. When you collapse this region, you can easily figure out what the collapsed code block does by looking at the name associated with the region. These directives are only useful in the visual designers such as Visual Studio .NET, for effective presentation of your code. When code is compiled, these directives are not present in the executable code.

WARNING

Windows Form Designer Generated Code The code enclosed in the code block titled Windows Form Designer Generated Code is required for Windows Forms Designer support, and you should not generally modify it.

If you collapse the Windows Form Designer Generated Code block and look at the other code that is present in the code view, you can see that almost all the code, other than a few `using` directives at the top, is enclosed in a namespace (I talk about `using` directives a bit later in this chapter). Using namespaces is a good practice because it helps you organize classes and other programming elements. Visual Studio .NET automatically organizes the classes for a new form in a namespace whose name is the same as the name of the project. (Because the project name in this case starts with a digit, Visual Studio .NET adds an underscore [`_`] at the beginning to make it a valid identifier.) When you create a Windows form using Visual Studio .NET, Visual Studio .NET defines a class that inherits its functionality from the standard `Form` class in the `System.Windows.Forms` namespace. In Step by Step 1.1, although you change the name of the code file containing the class from `Form1.cs` to `StepByStep1_1.cs`, the name of the class itself is not changed. Here's the class definition:

```
public class Form1 : System.Windows.Forms.Form
{
    //Form implementation goes here
}
```

`Form1` is the classname that Visual Studio .NET automatically generates for you when you create a Windows application. If you want to change it, you can either change it right in the code or you can modify the `Name` property of the form in the design view.

The `: System.Windows.Forms.Form` part of the code specifies that the `Form1` class inherits from the `Form` class that belongs to the `System.Windows.Forms` namespace. All the basic functionality of the `Form1` class (such as moving and resizing) comes from the base `Form` class.

NOTE

Names Can Differ It's a good convention to keep the same names for both the form's class and the file that contains the class definition, but in the .NET Framework, it's not a law that you do so.

NOTE

Static Constructors A class can have a *static constructor*, which is called automatically before any of the members of the class are accessed. A common use of static constructors is to initialize static fields and properties of the class.

Any class can have a constructor definition. A *constructor* is a method that is used to create new instances of a class. You can easily recognize a constructor because it has the same name as the class and is defined with syntax similar to that of a method definition, but it has no return type. Here's the constructor for the `Form1` class:

```
public Form1()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // TODO: Add any constructor code
    //after InitializeComponent call
}
```

Visual Studio .NET ignores the lines that start with `//` (they are comments and do not generate any code). The Windows Forms Designer puts just one line of code inside the default `Form1` constructor: a call to the `InitializeComponent()` method of the class. The Windows Forms Designer uses the `InitializeComponent()` method for storing the customizations done to the form through the design view. This method is defined in the Windows Form Designer Generated Code region. When you expand this region, you see code similar to this:

```
private void InitializeComponent()
{
    //
    // Form1
    //
    this.AutoScaleBaseSize =
        new System.Drawing.Size(5, 13);
    this.ClientSize =
        new System.Drawing.Size(292, 266);
    this.Name = "Form1";
    this.Text = "StepByStep1_1";
}
```

You can see here that the `Text` property of `Form1`, which you manipulate by using the Properties window, has been inserted as a code statement. Note the use of the `this` keyword to qualify the property names. The `this` keyword refers to the current instance of the class for which the method is called.

The next piece of code after the form's constructor is the `Dispose()` method. In its simple form it looks like this:

```
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}
```

The `Dispose()` method is an ideal place to put any cleanup code that you would like to be executed when you're done with the class.

NOTE

Calling `Dispose()` The CLR features automatic garbage collection. All memory resources that are no longer required are automatically garbage collected. `Dispose()` is not necessary for managed code, but it is a good place for cleanup code for any of the non-managed or non-memory resources that you create in a program.

Running a Windows Form

Finally in our code exploration comes the method that is making everything happen: the `Main()` method. Visual Studio .NET automatically generates the `Main()` method for you in the form's code. In this particular case, `Main()` is generated because you specified that the project should be created as a Windows form application. An application must have an execution starting point that is defined by the `Main()` method. When you execute the form, `Main()` is the method that receives the control first.

The `Main()` method for our simple Windows form example looks like this:

```
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}
```

The first line in this code is an attribute associated with the `Main()` method. Attributes are used to specify runtime behavior of a code element. The `STAThread` attribute specifies that the default threading model for this application is Single-Threaded Apartment (STA). It's a good idea to use this attribute with the `Main()` method because it is used when your application participates in Component Object Model (COM)-interoperability or does anything that requires Object Linking and Embedding, such as drag-and-drop or Clipboard operations.

The `Main()` method has a single line of code that invokes the `Run()` method of the `Application` class. The Application class provides methods and properties for managing a Windows application. The `Run()` method starts the application by creating the specified form onscreen and sends the application into a message loop. The application stays in the loop and responds to user messages (generated by such actions as moving or resizing the form) until the message loop is terminated because the form is closed.

WARNING

Keeping Things Synchronized If you modify the name of a form, either by using the code view or by using the Windows Forms Designer via the `Name` property, Visual Studio .NET does not automatically change the name of the form in the `Application.Run()` method. You have to change it manually.

Using `Form1()` inside the `Application.Run` statement calls the Form1 constructor, to return an instance of the newly created form. This displays the form when the application starts.

Any form that will initiate a Windows application by launching itself should have a `Main()` method, similar to the one discussed previously, defined in it. Normally only one form in a project (the form that acts as the starting form) has a `Main()` method.

If you refer to the .NET FCL documentation, you will see that the `Application` class belongs to the `System.Windows.Forms` namespace. But rather than uniquely referring to it as `System.Windows.Forms.Application`, the code refers to it as just `Application`. How is this possible? The language designers noted that typing the full namespace with a class every time it is used is a lot of typing, so they provided a shortcut for this via the `using` directive. Near the beginning of a program file, Visual Studio .NET typically includes the following `using` statements for a Windows application:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
```

Inclusion of these `using` directives tells the C# compiler to look for each class you are using in the namespaces specified in the `using` statement. The compiler looks up each namespace one-by-one, and when it finds the given class in one of the namespaces, it internally replaces the reference of the class with `NamespaceName.ClassName` in the code.

What happens when code uses two classes that have the same name but belong to different namespaces? The usage of the `using` directive discussed so far can't handle that situation; fortunately, there is another way you can use `using` directives. You can create aliases for a namespaces with the `using` directive. These aliases save you typing and qualify classes appropriately. Here is an example:

```
//Create namespace alias here
using Q316 = QueCertifications.Exam70316;
using Q306 = QueCertifications.Exam70315;

//use aliases to distinctly refer to classes
Q316.ExamQuestions.Display();
Q306.ExamQuestions.Display();
```

When you instruct Visual Studio .NET to run the form, it first compiles the form's code, using an appropriate language compiler (C# in this case for a C# Windows application). If there are no errors at compile time, the compiler generates an executable file with the name of the project and the extension `.exe` (for example, `316C01.exe`). The default location of the file is the bin\debug directory in the project's directory. You can explore where this file is located in the project through the Solution Explorer by following the steps in Step by Step 1.2.

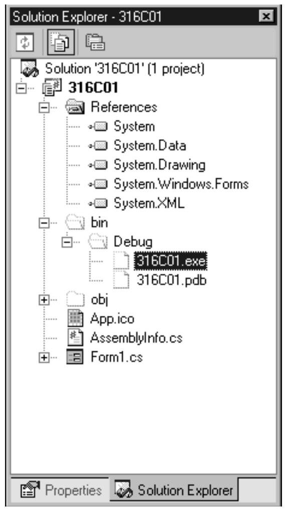


FIGURE 1.8
You can show all files in the Solution Explorer.

STEP BY STEP

1.2 Using the Solution Explorer to See Hidden Files

1. Open the Solution Explorer window if it is not already open by selecting View, Solution Explorer.
2. Click on the Project name `316C01` to select it. From the toolbar in the Solution Explorer, click the Show All Files icon.
3. A dimmed folder icon named `bin` becomes visible in the project's file hierarchy. Expand this folder fully by clicking the + signs (see Figure 1.8). You should see a file named `316C01.exe`. This is the project's executable file. The other file, named `316C01.pdb` (`pdb` stands for *program database*), stores debugging and project information.
4. Click the project name again to select it. From the toolbar in the Solution Explorer window click Show All Files. The `bin` folder is now hidden.

NOTE

Don't Confuse Library Names with Namespaces Sometimes the name of a library may look similar to the name of a namespace. For example, earlier in this chapter I talked about a namespace `System.Windows.Forms`, and there is a library by the same name that exists as `System.Windows.Forms.dll`. Don't be mistaken: They are totally different concepts. A library exists as a file and can contain code for one or more classes. Those classes may belong to different namespaces. A namespace is a logical organization of classes and has no physical form. A library is a physical unit that stores classes as a single deployment unit. Creation of code libraries is discussed in more detail in Chapter 4.

You might be wondering whether the little code that you are seeing in the form is enough for all of its functionality. Where is the code to actually draw the form onscreen? Where is the code that responds to dragging or resizing of the form? This form is inheriting its functionality from other classes. But where is the code for those classes? Code for various classes in the FCL is packaged as libraries (that is, `.dll` files), and Visual Studio .NET is smart enough to automatically include references to them in your project. It selects a few common libraries, depending on the project type, and it lets you include references to other libraries. You can see what libraries are included with a project by opening the Solution Explorer and navigating to the References hierarchy within the project (refer to Figure 1.8).

Using the `Application` Class

The `Application` class is responsible for managing a Windows Application. It provides a set of properties to get information about the current application (see Table 1.1). It also provides methods to start an application, end an application, and process the Windows messages (see Table 1.2). It is important to note here that all methods and properties of the `Application` class are static.

Because they're static, you need not create an instance of the `Application` class in order to use them. You can directly call these methods by prefixing them with the name of the class. As a matter of fact, creating an instance of the `Application` class is not possible. The class designers assigned a private access modifier to the constructor of the `Application` class to prevent you from creating instances of the class. If a constructor is not accessible for a class, it cannot be instantiated.

TABLE 1.1

SOME IMPORTANT STATIC PROPERTIES OF THE `Application` CLASS

Property Name	*Description*
`CompanyName`	Specifies the company name associated with the application
`CurrentCulture`	Specifies the culture information for the current thread
`CurrentInputLanguage`	Specifies the current input language for the current thread
`ExecutablePath`	Specifies the path of the executable file that started the application
`ProductName`	Specifies the product name for the current application
`ProductVersion`	Specifies the product version for the current application

TABLE 1.2

SOME IMPORTANT STATIC METHODS OF THE `Application` CLASS

Method Name	*Description*
`DoEvents()`	Processes all Windows messages currently in the message queue
`Exit()`	Terminates the application
`ExitThread()`	Exits the message loop on the current thread and closes all windows on the thread
`Run()`	Begins a standard application message loop on the current thread

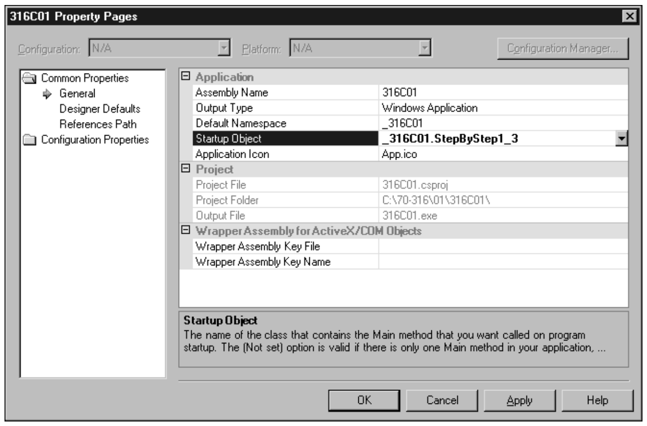
STEP BY STEP

1.3 Using `Application` Class Properties

1. Launch Visual Studio .NET. Select File, Open, Project. Navigate to the existing project `316C01` and open it.
2. In the Solution Explorer, right-click the project name and select Add, Add Windows Form. (Alternatively, you can do this by selecting Project, Add Windows Form). Name the new form `StepByStep1_3.cs`.
3. Switch to the code view, and just after the Windows Form Designer Generated Code region, insert the following `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_3());
    //Display a MessageBox
    MessageBox.Show(Application.ExecutablePath,
      "Location of Executable", MessageBoxButtons.OK,
           MessageBoxIcon.Information);
}
```

4. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, set the startup object to `_316C01.StepByStep1_3` (see Figure 1.9). Click OK to close the Property Pages window.

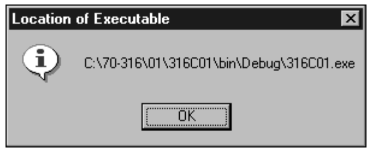


Application	
Assembly Name	316C01
Output Type	Windows Application
Default Namespace	_316C01
Startup Object	**_316C01.StepByStep1_3**
Application Icon	App.ico
Project	
Project File	316C01.csproj
Project Folder	C:\70-316\01\316C01\
Output File	316C01.exe
Wrapper Assembly for ActiveX/COM Objects	
Wrapper Assembly Key File	
Wrapper Assembly Key Name	

Startup Object
The name of the class that contains the Main method that you want called on program startup. The (Not set) option is valid if there is only one Main method in your application, ...

FIGURE 1.9
You can change the startup object by using the Property Pages window.

5. Select Debug, Run. The application displays the new form. When you close the form, the application displays a message box that shows the application's executable path (see Figure 1.10).


Location of Executable

C:\70-316\01\316C01\bin\Debug\316C01.exe

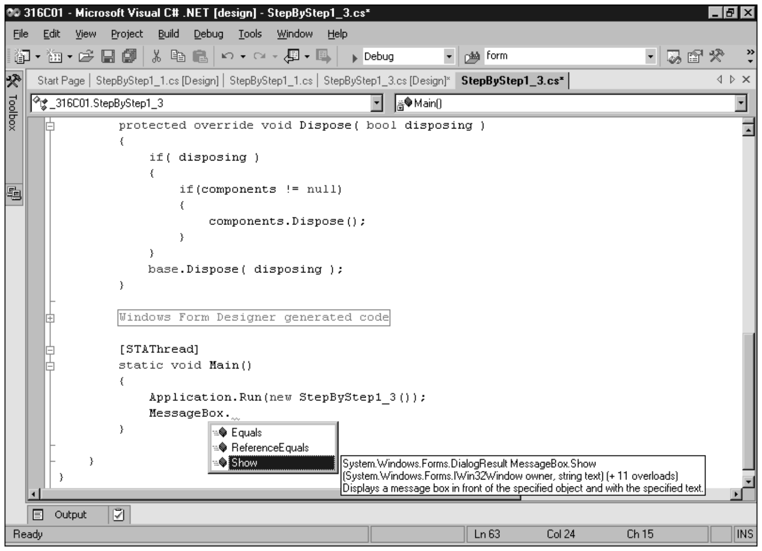
FIGURE 1.10
You can display an executable's path by using the `Application.ExecutablePath` property.

Because in Step by Step 1.3 you add another form to an existing Windows form project, the `Main()` method that starts the application is not automatically inserted in the form. The newly created form has its `Name` property properly set to `StepByStep1_3` because you explicitly specify the classname at the time of form creation. In Step by Step 1.3 you want to run the newly created form when the application starts. To accomplish this, you can manually add a `Main()` method in code and designate the newly added form as the startup object. Even though the project then contains two forms, each with a `Main()` method, only the `Main()` method of the startup object is executed when you start the project. The other form (`StepByStep1_1`) just exists there, doing nothing.

When you run the `project`, the form `StepByStep1_3` is displayed. When you close the form, it internally sends an exit message to the active application, which takes the control out of the `Application.Run()` method; the message box gets a chance to display itself.

`StepByStep1_3` uses the `Show()` method of the `MessageBox` class to display a message box to the user. The message box displays the value of the `ExecutablePath` property of the `Application` class.

Because you're calling the `Show()` method prefixed by the name of the class (`MessageBox`), `Show()` must be a static method. Note that as you type the code, Visual Studio .NET helps you with syntax, methods, and properties of various code items, by using a feature called IntelliSense (see Figure 1.11).


316C01 - Microsoft Visual C# .NET [design] - StepByStep1_3.cs*
File Edit View Project Build Debug Tools Window Help
Debug
form
Start Page | StepByStep1_1.cs [Design] | StepByStep1_1.cs | StepByStep1_3.cs [Design]* | StepByStep1_3.cs*
_316C01.StepByStep1_3
Main()
Toolbox
Equals
ReferenceEquals
Show
Output
Ready
Ln 63
Col 24
Ch 15
INS

```
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if(components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        Windows Form Designer generated code

        [STAThread]
        static void Main()
        {
            Application.Run(new StepByStep1_3());
            MessageBox.
        }
    }
}
```

System.Windows.Forms.DialogResult MessageBox.Show
(System.Windows.Forms.IWin32Window owner, string text) (+ 11 overloads)
Displays a message box in front of the specified object and with the specified text.

FIGURE 1.11
IntelliSense helps you easily complete statements.

As a quick exercise, try placing the `MessageBox.Show()` method before `Application.Run()`, and note the difference in execution.

Using the `MessageBox` Class

The `MessageBox` class belongs to the `System.Windows.Forms` namespace and inherits from the `Object` class. In addition to the methods that it has as a result of its inheritance from `Object`, it provides a `Show()` method that you can use in 12 different variations (as you can see with the help of IntelliSense) to display different kinds of message boxes to the user. Some of the arguments of the `Show()` method are values of enumeration types such as `MessageBoxButtons` and `MessageBoxIcons`. *Enumeration types* (also known as *enums*) provide a set of named constants called the enumerator list. You will see an extensive usage of enumeration types in .NET FCLs. They are the preferred, type-safe, and object-oriented way of referring to a list of constant values. For example, the `MessageBoxButtons` enumeration type provides a list of enumerators. Each of these values specifies a set of buttons that can be displayed in a message box. Table 1.3 lists the values of the `MessageBoxButtons` enumeration type and their meanings.

TABLE 1.3

`MessageBoxButtons` ENUMERATORS

Enumerator Value	*The Message Box Shows These Buttons*
`AbortRetryIgnore`	Abort, Retry, and Ignore
`OK`	OK
`OKCancel`	OK and Cancel
`RetryCancel`	Retry and Cancel
`YesNo`	Yes and No
`YesNoCancel`	Yes, No, and Cancel

NOTE **Using Message Boxes** Message boxes can help you quickly debug code in some instances. For example, you can use a message box to display the values of variables, fields, and properties at different stages in program execution.

The `MessageBoxIcons` enumeration determines the icon on a message box. Table 1.4 lists the values of the `MessageBoxIcons` enumerators and their meanings. (Note that some of the names have identical meanings.)

TABLE 1.4

`MessageBoxIcons` ENUMERATORS

Enumerator Value	*The Message Box Contains This Symbol*
`Asterisk`	A lowercase letter *i* in a circle
`Error`	A white *X* in a circle with a red background
`Exclamation`	An exclamation point in a triangle with a yellow background
`Hand`	A white *X* in a circle with a red background
`Information`	A lowercase letter *i* in a circle
`None`	No symbol
`Question`	A question mark in a circle
`Stop`	A white *X* in a circle with a red background
`Warning`	An exclamation point in a triangle with a yellow background

REVIEW BREAK

- Each form definition is contained in a class. Forms generally derive their functionality from the standard `System.Windows.Forms.Form` class.
- The Solution Explorer window allows you to manage all the files in a project.
- The `Main()` method acts as an entry point for a class. Execution of code inside a class begins from the `Main()` method.
- The `Application.Run()` method sends a form to a message loop that allows the form to listen to user interactions until the form is closed.
- The `MessageBox` class can be used to display various types of messages to the user.

SETTING AND ADDING PROPERTIES TO A WINDOWS FORM

Create a Windows form by using the Windows Forms Designer:

- **Add and set properties on a Windows form.**

The properties of an object provide a mechanism through which objects can expose their characteristics to the external world. In the following sections you will learn how to customize a form's appearance by using its properties. You will also learn how you can add your own properties to a form.

Using the Visual Designer to Set Windows Form Properties

A Windows form derives from several other classes, such as `Object`, `Control`, and so on, through a chain of inheritances.

As a result of this inheritance, the Windows form inherits the properties of its parent classes in addition of its own specific properties. All these properties are available for easy manipulation through the Properties window in Visual Studio .NET. Step by Step 1.4 shows you how to manipulate some of these properties to get a feel for how they affect the behavior of a form.

STEP BY STEP

1.4 Working with Windows Form Properties

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_4`.
2. Right-click the form and select Properties from the context menu. Find the `BackColor` property and click the down arrow of the drop-down list. This invokes a tabbed list of colors in which three categories of colors are available: Custom, Web, and System. Click the tab titled Web and select `AntiqueWhite` from the list. Note that the form's surface immediately changes color to reflect this property change.
3. In the Properties window, locate the property named `FormBorderStyle` and change its value from `Sizable` to `FixedSingle`.
4. In the Properties window, look for the property named `Size`. Expand this property by clicking the + sign next to it. Change the `Width` subproperty to `400` and the `Height` subproperty to `200`.
5. Go to the `StartPosition` property in the list of properties and change its value to `CenterScreen`.
6. Change the `MinimizeBox` property to `False`.
7. Right-click anywhere on the form and select View Code from the context menu. In the code view, insert the following code just after the Windows Form Designer Generated Code region:

continues

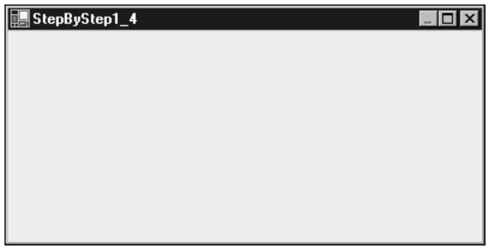


FIGURE 1.12
You can change a form's properties—such as `BackColor`, `Size`, and `FormBorderStyle`—to customize its appearance.

continued

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_4());
}
```

8. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, set `_316C01.StepByStep1_4` as the startup object. Click OK to close the Property Pages window.
9. Select Debug, Run. The application displays the form, showing the effect of the property settings that you made. The result looks similar to that shown in Figure 1.12.

In Step by Step 1.4, you manipulate various properties of a form to change its visual appearance. When you invoke the Properties window for the form, you see a big list of properties that are available to you. These properties provide significant control over the characteristics of the form. Table 1.5 lists some of the important properties of the `Form` class.

TABLE 1.5

SOME IMPORTANT PROPERTIES OF THE `Form` CLASS

Property Name	*Description*
`BackColor`	Specifies the background color of the form
`BackgroundImage`	Specifies the background image displayed in the form
`ClientSize`	Specifies the size of the client area of the form
`ControlBox`	Indicates whether a control box needs to be displayed in the caption bar of the form
`DesktopLocation`	Specifies the location of the form on the Windows desktop
`Enabled`	Indicates whether a control can respond to user interaction
`FormBorderStyle`	Specifies the border style of the form
`Handle`	Gets the Window Handle (HWND) of the form
`HelpButton`	Indicates whether a Help button is to be displayed on the caption bar of the form

Property Name	*Description*
`Icon`	Specifies the icon for the form
`MaximizeBox`	Indicates whether a maximize button is to be displayed on the caption bar of the form
`MaximumSize`	Specifies the maximum size to which the form can be resized
`MinimizeBox`	Indicates whether a minimize button is to be displayed in the caption bar of the form
`MinimumSize`	Specifies the minimum size to which the form can be resized
`Modal`	Indicates whether the form is to be displayed modally
`Name`	Specifies the name of the form
`Opacity`	Specifies the opacity level of the form
`ShowInTaskbar`	Indicates whether the form is to be displayed in the Windows taskbar
`Size`	Specifies the size of the form
`StartPosition`	Specifies the starting position of the form at runtime
`TopMost`	Indicates whether the form should be displayed as the top-most form of the application

The properties in Table 1.5 have various data types. You will find that some properties, such as `FormBorderStyle`, are enumeration types; some properties, such as `Size`, are of type struct, and their values are determined by the values of their contained members (for example, `X` and `Y`); other properties, such as `MinimizeBox`, accept a simple Boolean value. The Properties window provides a nice user interface for working with these values.

To increase your understanding about what's going on behind the scenes, it would be a good idea to switch to the code view, expand the Windows Form Designer Generated Code region, and analyze the generated code.

Setting Windows Form Properties Programmatically

Using the Windows Forms Designer is a quick and easy way to manipulate control properties, but the designer can only set the properties at design time. What would you do if wanted to change the appearance of a form at runtime? You could write your own code in the code view.

In complex projects, as you need more functionality in an application, you will find yourself switching frequently to the code view. Sometimes you can learn a lot about the .NET Framework by using the code view because it gives you an opportunity to directly play with .NET Framework data structures.

Step by Step 1.5 lets you explore some more properties of a Windows form and programmatically code the properties that are used in Step by Step 1.4. Step by Step 1.5 also shows you how to create a form programmatically.

STEP BY STEP

1.5 Setting Windows Form Properties Programmatically

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_5`.

2. Right-click anywhere on the form and select View Code from the context menu. In the code view, insert the following code just after the Windows Form Designer Generated Code region:

```
[STAThread]
static void Main()
{
    // Create StepByStep1_5 object
    // and set its properties
    StepByStep1_5 frm1_5 = new StepByStep1_5();
    frm1_5.BackColor = Color.AntiqueWhite;
    frm1_5.FormBorderStyle =
        FormBorderStyle.FixedSingle;
    frm1_5.Size = new Size(400,200);
    frm1_5.StartPosition =
       FormStartPosition.CenterScreen;
    frm1_5.MinimizeBox = false;
    Application.Run(frm1_5);
}
```

3. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, select `_316C01.StepByStep1_5` as the startup object. Click OK to close the Property Pages window.

4. Select Debug, Run. The form that is displayed looks similar to the form that you created in Step by Step 1.4 (refer to Figure 1.12).

In Step by Step 1.5, when you add a new Windows form to a project, a new class representing that form is created. As you can see in the code in Step by Step 1.5, you first create an object of the class so that you can later modify the object's properties. The modified form object is then passed as a parameter to the `Application.Run()` method that invokes the form.

While you are typing code in the code view, you see Visual Studio .NET helping you with the properties and syntaxes via IntelliSense. You should also note that values for properties such as `BackColor` and `FormBorderStyle` are encapsulated in enumerated types. You can find out what enumerated type to use for a property by hovering your mouse pointer over a property name in the code view. This displays a ToolTip that helps you identify the type of a property.

If you compare the code that you write manually in Step by Step 1.5 with the code that is generated by the Windows Forms Designer in Step by Step 1.4, you should see that the most significant difference is that the designer includes all its code for setting form properties in the `InitializeComponent()` method.

You can alternatively place the code inside the form's constructor after the call to the `InitializeComponent()` method. Including the code in the constructor ensures that the code is executed every time an instance of the form is created. The effect in this case would be the same as the effect of placing the code in the `InitializeComponent()` method.

So far you have seen how to get and set properties for a Windows form. The `Form` class also provides a set of methods, and Step by Step 1.6 demonstrates how to use them.

NOTE

Leaving the Autogenerated Code Alone The Windows Forms Designer manages the `InitializeComponent()` method. Putting your own code within that method might interfere with the designer's working. It is therefore generally recommended that you avoid modifying this method.

STEP BY STEP

1.6 Invoking Methods of Windows Forms

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_6`.

2. Right-click anywhere on the form and select View Code from the context menu. In the code view, insert the following code just after the Windows Form Designer Generated Code region:

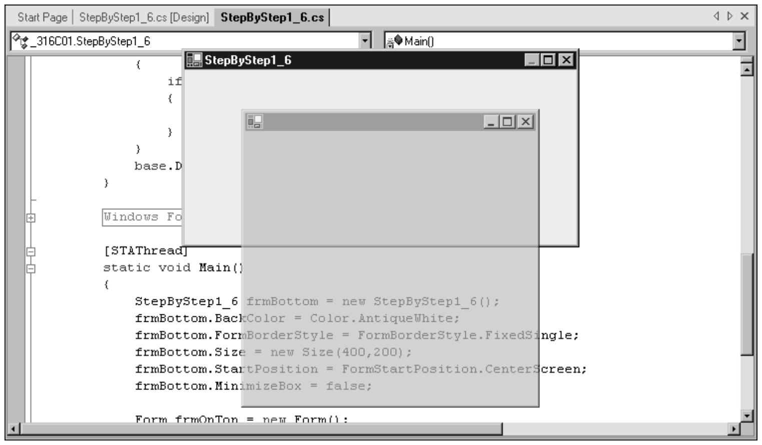
```
[STAThread]
static void Main()
{
    // Create StepByStep1_6 object
    // and set its properties
    StepByStep1_6 frmBottom = new StepByStep1_6();
    frmBottom.BackColor = Color.AntiqueWhite;
    frmBottom.FormBorderStyle =
        FormBorderStyle.FixedSingle;
    frmBottom.Size = new Size(400,200);
    frmBottom.StartPosition =
       FormStartPosition.CenterScreen;
    frmBottom.MinimizeBox = false;

    // Create a new form and set its
    // properties to stay on top
    Form frmOnTop = new Form();
    frmOnTop.TopMost = true;
    frmOnTop.Opacity = 0.7;
    frmOnTop.Show();

    Application.Run(frmBottom);
}
```

3. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, select `_316C01.StepByStep1_6` as the startup object. Click OK to close the Property Pages window.

4. Select Debug, Run. The application displays two forms. The top form is transparent and stays on the top, even when you click the other form (see Figure 1.13).

NOTE

The `Opacity` Property Transparent forms are supported only on operating systems that can display layered windows. Such operating systems include Windows 2000, Windows XP and later versions of Windows. The `Opacity` property has no effect when you run a program on older operating systems such as Windows 98.



```
            {
                if
                {

                }
            }
            base.D
        }

        [STAThread]
        static void Main()
        {
            StepByStep1_6 frmBottom = new StepByStep1_6();
            frmBottom.BackColor = Color.AntiqueWhite;
            frmBottom.FormBorderStyle = FormBorderStyle.FixedSingle;
            frmBottom.Size = new Size(400,200);
            frmBottom.StartPosition = FormStartPosition.CenterScreen;
            frmBottom.MinimizeBox = false;

            Form frmOnTop = new Form();
```

FIGURE 1.13
The `Show()` method sets the `Visible` property of a form to `true`.

In Step by Step 1.6 you create a new form by creating an instance of the `Form` class. You set the properties of the new form so that it is the topmost form in the application and then you reduce the opacity to make the form slightly transparent. Finally, you invoke the `Show()` method of the form to actually display the form onscreen.

There are two forms in Step by Step 1.6, but you are calling the `Show()` method for only one of them. When you run the program, you actually see both forms onscreen. How? It happens because `frmBottom` is passed as a parameter to the `Application.Run()` method. When the `Application.Run()` method starts the application by launching `frmBottom` onscreen, in the process it internally calls the `Show()` method for `frmBottom`. In the running program, if you close `frmOnTop`, you close just that form, but if you close `frmBottom`, you close all open forms and quit the application.

EXAM TIP

`Form.Close()` Versus `Form.Hide()` When you close a form by using its `Close()` method, all resources related to that form are released. You cannot call the `Show()` method to make the form visible again because the form itself does not exist anymore. If you want to temporarily hide a form and show it at a later time, you can use the form's `Hide()` method. Using the `Hide()` method is equivalent to setting the form's `Visible` property to `false`. The form still exists in memory, and you can make it visible any time by calling the `Show()` method of the form or by setting the form's `Visible` property to `true`.

GUIDED PRACTICE EXERCISE 1.1

You are a Windows developer for SpiderWare, Inc. In one of your applications you are required to create a form that enables users to set various options of the application. You want the form to have the following characteristics:

- It should have a thin title bar showing the text Options and a close button. The user should not be able to resize, minimize, or maximize this form.

continues

continued

- It should always appear on top of the other forms in the application.
- It should be always displayed in the center of the screen.

How would you create such a form?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Open the project `316C01`. Add a Windows form with the name `GuidedPracticeExercise1_1` to the project.
2. Open the Properties window for the form. Change the value of the `Text` property to `Options`.
3. Change the `FormBorderStyle` property to `FixedToolWindow`.
4. Change the `StartPosition` property to `CenterScreen`.
5. Change the `TopMost` property to `true`.
6. Switch to the code view, and insert the following `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new GuidedPracticeExercise1_1());
}
```

7. Set the form as the startup object and execute the program.

If you have difficulty following this exercise, review the section "Designing a Windows Form by Using Windows Forms Designer," earlier in this chapter. Also spend some time looking at the various properties that are available for a form in the Properties window. Experimenting with them in addition to reading the text and examples in this chapter should help you relearn this material. After doing that review, try this exercise again.

Adding New Properties to a Windows Form

In addition to using the existing properties, you can add custom properties to a Windows form. You can use a custom property to store application-specific data.

STEP BY STEP

1.7 Adding New Properties to a Windows Form

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_7`.

2. Right-click anywhere on the form and select View Code from the context menu. In the code view, insert the following code just after the Windows Form Designer Generated Code region:

```
//define constant values for State
public enum State{Idle, Connecting, Processing}

//use this field as storage location
//for FormState property
private State formState;

//set attributes for FormState property
[Browsable(true),
EditorBrowsable(EditorBrowsableState.Never),
Description("Sets the custom Form state"),
Category("Custom")]

//Creating FormState property
public State FormState
{
    get
    {
        return formState;
    }
    set
    {
        formState = value;
        switch (formState)
```

continues

continued

```
        {
            case State.Idle:
                this.BackColor = Color.Red;
                this.Text = "Idle";
                break;
            case State.Connecting:
                this.BackColor = Color.Orange;
                this.Text = "Connecting...";
                break;
            case State.Processing:
                this.BackColor = Color.Green;
                this.Text = "Processing";
                break;
        }
    }
}
```

3. Change the form's constructor so that it looks like this:

```
public StepByStep1_7()
{
    // Default code of the Constructor
    //set the FormState property of this form
    this.FormState = State.Processing;
}
```

4. Add the following `Main()` method to the form:

```
[STAThread]
public static void Main()
{
    Application.Run(new StepByStep1_7());
}
```

5. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, select `_316C01.StepByStep1_7` as the startup object. Click OK to close the Property Pages window.

6. Select Debug, Run. The project shows a green-colored form onscreen.

The most important thing to note in Step by Step 1.7 is that you add a custom property named `FormState` to a form. The property that you create is even visible in the IntelliSense list when you try to access the members of this form object by typing a period (.) after the form's name in the form's constructor code.

To a program that uses the properties, the properties in Step by Step 1.7 look just like data fields, but properties by themselves do not have any storage locations, and their definitions look almost like method definitions. Properties provide two accessors (`get` and `set`) that would be called when a program would like to read from or write to the property. In Step by Step 1.7 you use a private field `formState` that works as a storage location for the `FormState` property; because `formState` is private, the rest of the world can access it only through the `FormState` public property. The data type of `FormState` is the enumerated type `State`, defined with a limited set of named constant values at the beginning of the code segment.

If you go to the design view and inspect the properties of form `StepByStep1_7` through Properties window, you will not find the `FormState` property listed along with other properties. This is because the Properties window shows only the properties of the base class. You can think of it like this: The Properties window helps you design a new class (`StepByStep1_7`) in terms of a class that already exists (`System.Windows.Forms.Form`). So while you are designing the `StepByStep1_7` class, if the Properties window would rather show the properties of the `StepByStep1_7` class, then it's similar to defining a class in terms of a class that is itself under construction.

If you instead define a form by using the completely created class `StepByStep1_7`, it would make sense to have the `FormState` property available in the derived form through the Properties window. (I talk about this in a moment.)

Note that the `FormState` property has a big list of attributes. Such a big list isn't required to create a property, but it helps you specify the runtime behavior of the property. Table 1.6 describes how attributes can control the behavior of a property.

EXAM TIP

Read-Only and Write-Only Properties The `get` and `set` accessors allow both read and write access to a property. If you want to make a property read-only, you should not include a set accessor in its property definition. On the other hand, if you want a write-only property, you should not include a get accessor in the property definition.

NOTE

Using the `get` and `set` Accessors Accessors of a property can contain executable statements. The code contained in the `get` accessor executes when a program reads the value of a property. Similarly, when a program writes a value to a property, it executes the `set` accessor of the property.

TABLE 1.6

ATTRIBUTES THAT CONTROL THE BEHAVIOR OF A PROPERTY

Attribute Name	*Description*
`Browsable`	Indicates whether the property is displayed in the Properties window. Its default value is true.

continues

TABLE 1.6 *continued*

ATTRIBUTES THAT CONTROL THE BEHAVIOR OF A PROPERTY

Attribute Name	*Description*
`EditorBrowsable`	Indicates whether the property should appear in the IntelliSense list of an object in the code view. Its value is of the `EditorBrowsableState` enumeration type, with three possible values—`Advanced`, `Always`, and `Never`. Its default value is `Always`, which means "always list this property." If you change the value to `Never`, the property is hidden from the IntelliSense feature.
`Description`	Specifies a description string for the property. When the `Description` property is specified, it is displayed in the description area of the Properties window when the property is selected.
`Category`	Specifies the category of the property. The category is used in the Properties window to categorize the property list.

You cannot see the effect of using the attributes listed in Table 1.6 in the form you have been creating. However, you would see their effects if you inherited a form from that form. You'll learn more about this in the next section of this chapter.

REVIEW BREAK

- Properties let you customize the appearance and behavior of a Windows form.
- The Windows Forms Designer lets you define a form based on the properties that are available in its base class (which is usually the `Form` class).
- You can add custom properties to a form.
- Attributes let you define the runtime behavior of a property.

USING VISUAL INHERITANCE

Create a Windows form by using the Windows Forms Designer:

- **Create a Windows form by using visual inheritance**

Earlier in this chapter, in the section “Understanding Classes, Inheritance, and Namespaces,” you learned that when a class inherits from its base class, it derives a basic set of functionality from the base class. No additional programming needs to be done in the derived class to enable that functionality. You are free to add extra functionality to the derived class to make it yet more useful.

Because a Windows form is a class, inheritance applies to it. In some cases you will find yourself creating new forms that are almost like ones you have previously created, but the new forms need some additional functionality. In such a case, rather than create a new form from scratch, you can inherit it from a form that has similar functionality and later customize the inherited form to add extra functionality to it.

When inheritance is applied to a Windows form, it is known as *visual inheritance* because it results in the inheritance of the visual characteristics of the form, such as its size, color, and any components placed on the form. You can also visually manipulate the properties that you inherit from the base class.

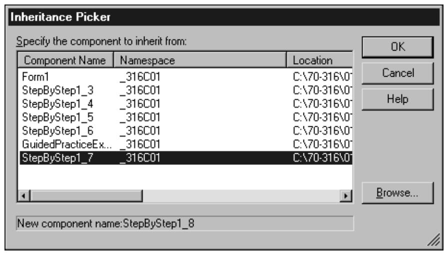
Step by Step 1.8 shows how to inherit from an existing form by using visual inheritance.

STEP BY STEP

1.8 Using Visual Inheritance

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Inherited Form from the context menu. Name the new form `StepByStep1_8` and click the Open button.
2. From the Inheritance Picker dialog box (see Figure 1.14), select the component named `StepByStep1_7` and click OK.

continues

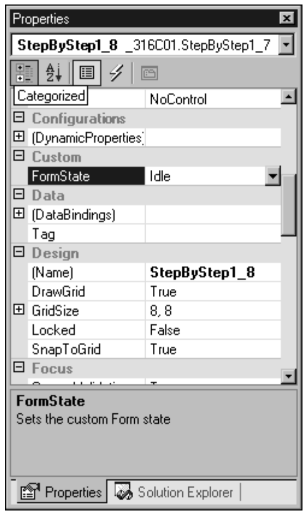


Specify the component to inherit from:

Component Name	Namespace	Location
Form1	_316C01	C:\70-316\0
StepByStep1_3	_316C01	C:\70-316\0
StepByStep1_4	_316C01	C:\70-316\0
StepByStep1_5	_316C01	C:\70-316\0
StepByStep1_6	_316C01	C:\70-316\0
GuidedPracticeEx...	_316C01	C:\70-316\0
StepByStep1_7	_316C01	C:\70-316\0

New component name:StepByStep1_8

FIGURE 1.14
The Inheritance Picker dialog box allows you to choose a base class for a form.


Properties

StepByStep1_8 _316C01.StepByStep1_7

Categorized	NoControl
⊟ Configurations	
⊞ (DynamicProperties)	
⊟ Custom	
FormState	Idle
⊟ Data	
⊞ (DataBindings)	
Tag	
⊟ Design	
(Name)	**StepByStep1_8**
DrawGrid	True
⊞ GridSize	8, 8
Locked	False
SnapToGrid	True
⊟ Focus	

FormState
Sets the custom Form state

Properties | Solution Explorer

FIGURE 1.15
The `FormState` property is available in the Properties window in the proper category, and it has the proper description.

continued

3. Open the Properties window. Click the Categorized icon on its toolbar. In the category Custom, change the `FormState` property within that category to `Idle` (see Figure 1.15).

4. Add the following `Main()` method to the form:

```
[STAThread]
public static void Main()
{
    Application.Run(new StepByStep1_8());
}
```

5. In the Solution Explorer, right-click the project name and select Properties from the context menu. In the Property Pages window, select `_316C01.StepByStep1_8` as the startup object. Click OK to close the Property Pages window.

6. Select Debug, Run. Because you have set the `FormState` property to `Idle`, the form displays in red.

EXAM TIP

Inheriting Private Members A derived class inherits all the members of a base class, but the private members of the base class are not accessible in the derived class because the variables with private modifiers in the base class are hidden from the derived classes.

When you run the form in Step by Step 1.8, note that the newly created form has the same behavior as the form created in Step by Step 1.7. The new form inherits its behavior from the form that already existed. In other words, the form named `StepByStep1_8` is based on the form named `StepByStep1_7`. You have access to all browsable properties of the base form through the Properties window. When you select the FormState property of the form, you have access to its possible values in a drop-down list. Because of the runtime attribute applied to the `FormState` property in the base class, this property is filed in the `Custom` category. You are able to see the description of this property at the bottom of the Properties window when the property is selected.

After you inherit a form, you can add extra functionality to it. This functionality is available only in the newly created class and the classes you later derive from it, but it does not affect any of the base classes.

GUIDED PRACTICE EXERCISE 1.2

You are a Windows developer for SpiderWare, Inc. In one of your applications, you recently created a form that will be used to set various options of the application (refer to Guided Practice Exercise 1.1). Your application now requires another form such as the one you designed earlier. However, the color of this form should always be the same as the color of the user's desktop.

How would you create such a form?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Open the project `316C01`. Add an inherited form with the name `GuidedPracticeExercise1_2` to this project.
2. From the Inheritance Picker dialog box select the component named `GuidedPracticeExercise1_1` and click OK.
3. Open the Properties window for the form, and change the `BackColor` property to `Desktop`.
4. Switch to the code view, and insert the following `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new GuidedPracticeExercise1_2());
}
```

5. Set the form as the startup object and execute the program.

If you have difficulty following this exercise, review the sections "Designing a Windows Form by Using Windows Forms Designer" and "Using Visual Inheritance." Make sure you work through Guided Practice Exercise 1 before you attempt this one. The text and examples presented in these sections should help you relearn this material. After doing that review, try this exercise again.

NOTE

The `EditorBrowsable` Bug If you are inheriting a form from the same project, even though you have set the `EditorBrowsable` attribute for a property to be `Never`, you can see this property via IntelliSense help in the derived form. This feature works fine with Visual Basic .NET but is a known bug with Visual C# .NET.

The `EditorBrowsable` attribute works fine, however, if you are visually inheriting from a form that is present in a separate assembly. You will learn more about creating assemblies in Chapter 4.

REVIEW BREAK

- Form inheritance allows you to create a new form by inheriting it from a base form. This allows you to reuse and extend earlier coding efforts.
- The Windows Forms Designer lets you visually inherit a form from an existing form through the Inheritance Picker dialog box. You can also visually manipulate the inherited properties through the Properties window.

EVENT HANDLING

Create, implement, and handle events.

When you perform an action with an object, the object in turn raises events in the application. Dragging the title bar of a form and moving it around, for example, generates an event; resizing a form generates another event. A large portion of code for any typical Windows application is the code that is responsible for handling various events that the application responds to. Events are at the heart of graphical user interface (GUI)–based programming. An *event handler* is a method that is executed as a response to an event.

Not all events are triggered by user actions. Events may be triggered by changes in the environment such as the arrival of an email message, modifications to a file, changes in the time and completion of program execution, and so on.

With C#, you can define your own custom events that a class will listen to (you'll see how to do this in Chapter 4). You can also handle an event by executing custom code when the event is fired. The following sections discuss two different ways to handle events:

- Handling events by attaching a delegate
- Handling events by overriding a protected method of a base class

Handling Events by Attaching a Delegate

When you create a Windows form, it inherits from the Form class. The Form class has a set of events that are already defined and that you can access through the Properties window (see Figure 1.16). If your program needs to take actions when one of those events is fired, it must define an appropriate event handler. The event handler must be registered with the event source so that when the event occurs, the handler will be invoked (this is also referred to as *event-wiring*). It is possible to have multiple event handlers interested in responding to an event. It is also possible to have a single event handler respond to multiple events.

The visual designer uses event wiring through delegates for handling events, so for most of the book it is also the preferred approach to event handling.

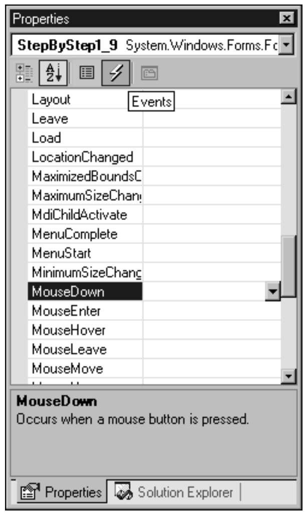


FIGURE 1.16
The Properties window lets you access events that are defined for an object.

STEP BY STEP

1.9 Handling the `MouseDown` Event

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_9` and click the Open button.
2. Open the Properties window of the form. Change the `Text` property to `Event Handling Form`.
3. In the Properties window, click the Events icon (which looks like a lightning bolt; refer to Figure 1.16) on the toolbar.
4. Look for an event named `MouseDown()`, and double-click the row containing the `MouseDown` event. This takes you to the code view, where Visual Studio .NET inserts a template for the `MouseDown` event handler. Add code to the event handler so that it looks like this:

continues

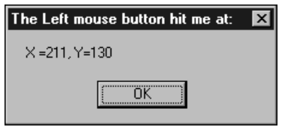
continued

```
private void StepByStep1_9_MouseDown(
 object sender, System.Windows.Forms.MouseEventArgs e)
{
    MessageBox.Show(
     String.Format("X ={0}, Y={1}", e.X, e.Y),
     String.Format("The {0} mouse button hit me at:",
      e.Button.ToString()));
}
```

5. Insert the following `Main()` method after the event handling code you just added:

```
 [STAThread]
static void Main()
{
    Application.Run(new StepByStep1_9());
}
```

6. Set the form as the startup object. Run the project. Observe that whenever you click the form area, a message box appears, displaying the position of the click and whether the left or right mouse button is pressed (see Figure 1.17).



```
The Left mouse button hit me at:
X =211, Y=130
```

FIGURE 1.17
The event handler displays a message box showing event-related data.

Step by Step 1.9 involves responding to the `MouseDown` event of a form. The form inherits the `MouseDown` event (and many others) from its base class Form. The Properties window lets you see all the events that are available for a control or another object. You choose `MouseDown` from that list, and when you double-click the event name, the designer switches to the code view and inserts a template for the `MouseDown` event handler. You insert a line of code that generates a message box, showing the coordinates of the point at which the mouse button is pressed. This information comes from the `MouseEventArgs` parameter, passed to the event handling method.

The name of the event handler is `StepByStep1_9_MouseDown` (that is, it uses the form *`ClassName_EventName`*). Visual Studio .NET follows this general naming convention for naming event handlers. An event handler normally has a void return type and accepts two arguments: the object on which the event occurred and an argument of type `EventArgs` (or a type derived from it, such as `MouseEventArgs`) that contains event-related data. In the `StepByStep1_9_MouseDown` event handler, the second argument is of type `MouseEventArgs`, and it contains data that is specific to mouse events (such as the position where the button was pressed).

Table 1.7 shows the kind of information that can be retrieved from the `MouseEventArgs` object. The type of the second argument depends on the nature of event. Visual Studio .NET automatically determines it for you, but if you write the event handler manually, you have to look in the documentation to find its correct type. The code inside the event handler is straightforward; it displays a message box that shows coordinates of the mouse location as well as which mouse button is pressed.

TABLE 1.7

`MouseEventArgs` PROPERTIES

Member	*Description*
`Button`	Returns a value of type `MouseButtons` that specifies which mouse button is pressed
`Clicks`	Returns the number of times the mouse button is pressed and released
`Delta`	Acts as a signed count of the number of detents (that is, notches of the mouse wheel) the mouse wheel has rotated
`X`	Specifies the x-coordinate of a mouse click
`Y`	Specifies the y-coordinate of a mouse click

How does the event handler get wired up with the actual event? The designer does it for you, when you double-click the event name in the Properties window. You can expand the designer-generated code to find the following line of code:

```
this.MouseDown +=
   new System.Windows.Forms.MouseEventHandler(
     this.StepByStep1_9_MouseDown);
```

This looks like a complex statement. If you break it down to understand it properly, however, you'll see that there are three parts to it:

- `MouseDown` is the name of an event.
- `MouseEventHandler` is the delegate of the `MouseDown` event.
- `StepByStep1_9_MouseDown` is the name of an event handler.

And here is the role each of them is playing:

- The `MouseDown` event is raised when a mouse button is pressed. A set of event handlers can be attached to this event.

When the event is fired, it invokes all the attached event handlers. An event handler can be attached to this event only through its delegate object.

- The delegate type of the `MouseDown` event is `MouseEventHandler`. You can add event handlers to a `MouseDown` event only by adding new instances of the delegate to it. A *delegate* is a special type that is capable of storing a reference to a method with a specific signature (that is, the arguments and return type). Because it stores references of methods, a delegate can also invoke the methods dynamically when the event occurs. In the Visual Studio .NET documentation, the definition of `MouseEventHandler` delegate looks like this:

```
public delegate void MouseEventHandler(
object sender, MouseEventArgs e);
```

  This means that `MouseEventHandler` is capable of storing references to any method whose return type is void and that accepts two arguments: the first one of type `System.Object` and the other one of type `MouseEventArgs`. The `StepByStep1_9_MouseDown` event handler signature matches the criteria of this delegate, and hence its reference can be stored in an instance of a delegate of type `MouseEventHandler`.

  When you have an instance of the `MouseEventHandler` delegate, you can attach it to the event by using the addition syntax. += is used in the example, so if any event handlers are already attached to this event by the base class, they remain in the list.

- `StepByStep1_9_MouseDown` is the name of the method that is responding to this event. The keyword qualifies it for the current instance of the form. When a method name is used alone, without any argument list, it works as a reference to the actual method definition. That reference is passed to the delegate. At a later stage, when the event occurs, that method is invoked through its reference that is maintained by the delegate object.

NOTE

Detaching an Event Handler You can detach an event handler from an event by using the -= syntax that is similar to the += syntax. Although detaching an event handler is generally not required, you might want to detach an event if at runtime, you are no longer interested in responding to a particular event.

EXAM TIP

Adding Versus Replacing Delegates When you are attaching delegates, you should usually use the += syntax rather than the = syntax. This is because = attaches the current delegate, and any other previously attached delegates are lost. Using += ensures that you preserve the list of previously attached event handlers.

To manually write code to handle an event, you would follow these steps:

1. Look up the Visual Studio .NET documentation to find the appropriate event for a class.
2. Find out the delegate type for this event.

3. Based on the delegate signature, create an event handler.
4. Create an instance of the event's delegate that contains a reference to the event handler method.
5. Add to the event the delegate instance from step 4.

From these steps, you can see that the designer takes a lot of details away from you, and what you are required to do is simply write the actual code that will be executed when the event occurs.

As mentioned earlier in this chapter, it is possible to attach multiple event handlers to an event. Step by Step 1.10 shows how to attach a second event handler to the `MouseDown` event from Step by Step 1.9. When you execute the code and press the mouse button on the form, both event handlers are executed.

STEP BY STEP

1.10 Attaching Multiple Event Handlers to the `MouseDown` Event

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_10` and click on the Open button.
2. Open the Properties window for the form. Change the form's `Text` property to `Multiple Events Handling Form`.
3. Search for the `MouseDown` event, and double-click on the row that contains the `MouseDown` event. Modify its event handler to look like this:

```
private void StepByStep1_10_MouseDown(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    MessageBox.Show(
      String.Format("X ={0}, Y={1}", e.X, e.Y),
      String.Format("The {0} mouse button hit me at:",
        e.Button.ToString()));
}
```

continues

continued

4. Switch to the design view, and look again for the `MouseDown` event in Properties window. If you again double-click the row that contains the `MouseDown` event, you see that a new event handler is not inserted. Therefore, you need to switch to the code view and add the following code for a second event handler:

```
private void SecondEventHandler(
  object sender, MouseEventArgs e)
{
    Form frm1_10 = (Form) sender;
    if (frm1_10.BackColor== Color.AntiqueWhite)
        frm1_10.BackColor = Color.LightCoral;
    else
        frm1_10.BackColor = Color.AntiqueWhite;
}
```

5. Insert the following `Main()` method after the event handling code from step 4:

```
[STAThread]
static void Main()
{
    StepByStep1_10 frm1_10= new StepByStep1_10();
    frm1_10.MouseDown += new MouseEventHandler(
      frm1_10.SecondEventHandler);
    Application.Run(frm1_10);
}
```

6. Set the form as the startup object. Run the project. Try clicking on the form with the left and right mouse buttons, and the form changes background color in addition to responding with a message box on every click.

In Step by Step 1.10 you add an event handler to the newly created instance of the form `StepByStep1_10`. The form then has two event handlers registered with the `MouseDown` event. The first event handler, inserted through the Properties window, is attached to the event when the `InitializeComponent` event is fired as part of the creation of a new instance of `StepByStep1_10`. The order in which both event handlers are executed is the order in which they were attached to the event.

Handling Events by Overriding a Protected Method of a Base Class

When you create a Windows form, it inherits from the `Form` class. By virtue of this inheritance, it has a set of public and protected methods and properties available to it from one of its base classes. The class view lets you see all the inherited members (see Figure 1.18). Some of the classes provide sets of protected methods that raise events. You can easily identify these classes because their naming convention is the word `On` followed by the name of the event. For example, `OnMouseDown()` is the protected method of the `Form` class that raises the `MouseDown` event. The `Form` class gets this protected method from the `Control` class.

The section “Handling Events by Attaching a Delegate” describes how to wire events through the use of delegate objects. An alternative way is to override the `On` method and write the event handling code there. Step by Step 1.11 demonstrates how to do that.

STEP BY STEP

1.11 Handling Events by Overriding the `OnMouseDown` Event

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_11` and click the Open button.
2. Open the Properties window of the form. Change the `Text` property of the form to `Event Handling through OnMouseDown`.
3. Open the class view by selecting View, Class View (or by pressing Ctrl+Shift+C). Navigate to the `StepByStep1_11` node. Expand the `Bases` and `Interfaces` node. You should see a node corresponding to the base class `Form`. Keep expanding until you see members of the `Control` class (see Figure 1.18).

continues

continued

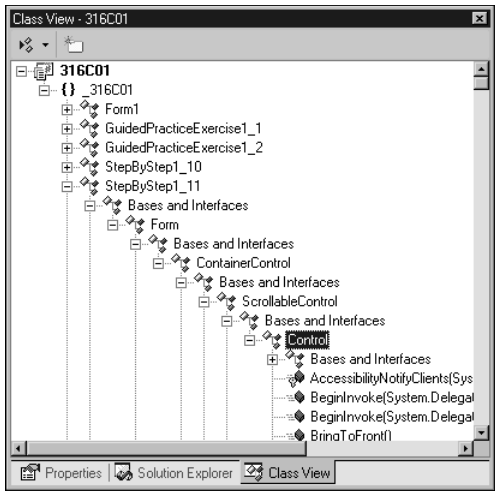


FIGURE 1.18
The class view lets you explore the complete inheritance hierarchy.

4. In the expanded tree, look for a method in the `Control` class named `OnMouseDown()`. Right-click the method name and select Add, Override from the context menu. This generates a template for the `OnMouseDown()` method in your program and switches to code view. Modify the `OnMouseDown()` method so that it looks like this:

```
protected override void OnMouseDown(
     System.Windows.Forms.MouseEventArgs e)
{
    MessageBox.Show(
      String.Format("X ={0}, Y={1}", e.X, e.Y),
      String.Format("The {0} mouse button hit me at:",
        e.Button.ToString()));
}
```

5. Enter the following code for the `Main()` method after the `OnMouseDown()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_11());
}
```

6. Set the form as the startup object. Run the project and click on the form surface with the left and right mouse buttons. You see the form respond to the `MouseDown` event by displaying a message box.

The result of Step by Step 1.11 is similar to the result of Step by Step 1.9, in which you handle an event by attaching a delegate to it. Only the implementation is different.

What makes the code in Step by Step 1.11 work? It works because the `OnMouseDown()` method (similar to other `On` methods) is the core method that is invoked when the mouse button is pressed. This method is responsible for notifying all registered objects about the `MouseDown` event. This method is not new; it was in place and working hard even when you were using the delegate-based event-handling scheme. How is that possible when you didn't code the method in your earlier programs? Recall from the section "Using Visual Inheritance" that the derived class inherits all members from its base classes in its inheritance tree. Step by Step 1.11 gets the `OnMouseDown` event from the `Control` class. In the base class `Control`, the `OnMouseDown()` method is declared as follows:

```
protected virtual void OnMouseDown(
  System.Windows.Forms.MouseEventArgs e);
```

Because the method is protected, it is available in all classes derived from `Control`, such as a `Form` and the delegate-based event-handling form `StepByStep1_9`. This method actually invokes the calls to delegates when the event takes place. The event-handling scheme discussed earlier keeps that fact hidden from you for the sake of simplicity.

The `virtual` modifier in the original declaration means that if the derived classes are not satisfied by the definition of this method in the original base class and if they need to extend it, they can do so by overriding its definition. That's what you do in Step by Step 1.11. When you override the method in a derived class, its base class version is not called. Instead, the overriding member in the most-derived class (which is `StepByStep1_11` in this case) is called. That's how the version of the method that you write in Step by Step 1.11 is executed when the mouse button is pressed.

The sections "Handling Events by Attaching a Delegate" and "Handling Events by Overriding a Protected Method of a Base Class" discuss two schemes for event handling. Can these schemes exist together? To answer this question, try Step by Step 1.12.

STEP BY STEP

1.12 Mixing the Two Event Handling Schemes

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add, Add Windows Form from the context menu. Name the new form `StepByStep1_12` and click the Open button.
2. Open the Properties window. Change the `Text` property of the form to `Mixing Event Handling Techniques`.
3. Search for the `MouseDown` event, and double-click the row that contains the `MouseDown` event. Modify its event handler to look like this:

```
private void StepByStep1_12_MouseDown(object sender,
    System.Windows.Forms.MouseEventArgs e)
{
    MessageBox.Show(
      String.Format("X ={0}, Y={1}", e.X, e.Y),
      String.Format("The {0} mouse button hit me at:",
        e.Button.ToString()));
}
```

4. Add the following code after the event handling code from step 3:

```
protected override void OnMouseDown(MouseEventArgs e)
{
    if (this.BackColor== Color.AntiqueWhite)
        this.BackColor = Color.LightCoral;
    else
        this.BackColor = Color.AntiqueWhite;
}
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_12());
}
```

5. Set the form as the startup object and execute it. When you click the form area, the background color changes but the message box is not displayed.

It seems that in Step by Step 1.12 you do not achieve quite what you wanted to achieve. Don’t the two event handling techniques coexist?

When you click the mouse button, only the event handler code written inside the `OnMouseDown()` method executes, and the other handler that is wired with the help of designer does not execute. That is because Step by Step 1.12 does not code the `OnMouseDown()` method properly. To understand the mistake, you need to take a look at how the `OnMouseDown()` method works in its original form, if you hadn't overridden it. Its base class version would look something like this:

```
public event MouseEventHandler MouseDown;
protected virtual void OnMouseDown(MouseEventArgs e)
{
    if (MouseDown != null)
     {
       //Invokes the delegates.
       MouseDown(this, e);
      }
   }
}
```

The `OnMouseDown()` method is invoked when the mouse button is pressed. In its code, it checks whether the associated `MouseDown` event has a delegate list associated with it. If the list is not empty, it raises the `MouseDown` event that actually fires all attached event handlers with the help of their respective delegate objects. (Recall from earlier discussions that a delegate object holds a reference to the method name and can invoke it dynamically.)

In Step by Step 1.12 , because you override the definition of the `OnMousedown()` method, its old base class logic that used to call other event handlers no longer executes. As a result, the events added through the delegate list are not executed. How can you fix the problem? The solution to this problem also teaches you a good programming practice: You should call the base class implementation of the protected `On` method whenever you override it. The modified `OnMouseDown()` method in Step by Step 1.12 would look like this:

```
protected override OnMouseDown(MouseEventArgs e)
{
    //fixes the problem by also calling base
    //class version of OnMouseDown method
    base.OnMouseDown(e);
    if (this.BackColor== Color.AntiqueWhite)
        this.BackColor = Color.LightCoral;
    else
        this.BackColor = Color.AntiqueWhite;}
```

This modification allows the base class implementation of the `OnMouseDown()` method to be executed; this is the method where the delegates are processed.

EXAM TIP

Overriding Protected Methods of a Base Class A derived class extends the functionality of its base class. It is generally a good idea to call the base class version of a method when you override a method in a derived class. That way, the derived class has at least the level of functionality offered by the base class. You can of course write more code in the overridden method to achieve extended functionality of the derived class.

On the other hand, if a derived class does not call the base class version of a method from an overridden method in derived class, you are not able to access all the functionality provided by the base class in the derived class.

REVIEW BREAK

- Events allow a program to respond to changes in the code's environment.
- Custom code can be executed when an event fires if the code is registered with the event. The pieces of code that respond to an event are called *event handlers*.
- Event handlers are registered with events through delegate objects.
- It is possible to respond to an event by overriding the `On` method corresponding to an event. When you use this method, you should be sure to call the corresponding `On` method for the base class so that you don't miss any of the event handlers registered through delegates when the event is raised.

Building Graphical Interface Elements by Using the System.Drawing Namespace

Create a Windows form by using the Windows Forms Designer:

- **Build graphical interface elements by using the `System.Drawing` namespace.**

The FCLs provide an advanced implementation of the Windows Graphics Design Interface (also known as GDI+). The GDI+ classes can be used to perform a variety of graphics-related tasks such as working with text, fonts, lines, shapes, and images. One of the main benefits of using GDI+ is that it allows you to work with Graphics objects without worrying about the specific details of the underlying platform. The GDI+ classes are distributed among four namespaces:

- `System.Drawing`
- `System.Drawing.Drawing2D`

- `System.Drawing.Imaging`
- `System.Drawing.Text`

All these classes reside in a file named `System.Drawing.dll`.

Understanding the `Graphics` Objects

The `Graphics` class is one of the most important classes in the `System.Drawing` namespace. It provides methods for doing various kinds of graphics manipulations. The `Graphics` class is a sealed class and cannot be further inherited (unlike the `Form` class, for example). The only way you can work with the `Graphics` class is through its instances (that is, `Graphics` objects). A `Graphics` object is a GDI+ drawing surface that you can manipulate by using the methods of the `Graphics` class.

When you look in the documentation of the `Graphics` class, you see that there is no constructor available for this class, and hence a `Graphics` object cannot be directly created. Despite this, there are at least four ways you can get a `Graphics` object:

- Through the `Graphics` property of the `PaintEventArgs` argument passed to the `Paint` event handler of a control or a form. The `Graphics` object thus received represents the drawing surface of the object that was the source of the event.
- By calling the `CreateGraphics()` method of a control or form.
- By calling the `Graphics.FromHwnd()` method and passing it the handle of the current form.
- By calling the static `Graphics.FromImage()` method. This method takes an image object and returns a `Graphics` object that corresponds to that image. You can then use this `Graphics` object to manipulate the image.

When you have a `Graphics` object available, you have access to a drawing surface. You can use this surface to draw lines, text, curves, shapes, and so on. But before you can draw, you must understand the Windows forms coordinate system, which is discussed in the following section.

Understanding the Windows Forms Coordinate System

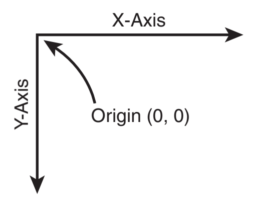


FIGURE 1.19
The Windows Forms Coordinate system is a two-dimensional coordinate system.

The Windows Forms library treats a Windows form as an object that has a two-dimensional coordinate system, as shown in Figure 1.19. Therefore, when you write text on the form or put controls on the form, the position is identified by a set of points. A *point* is a pair of numbers that is generally represented as (x, y) where x and y, respectively, denote horizontal and vertical distance from the origin of the form. The origin of the form is the top-left corner of the client area of the form. (The *client area* is the inner area of the form that you get after excluding the space occupied by title bar, sizing borders, and menu, if any.) The point of the origin is treated as (0, 0). The value of x increases to the right of the point of origin, and the value of y increases below the point of origin.

Two structures are available to represent points in a program—`Point` and `PointF`. These structures each represent an ordered pair of values (x and y). `Point` stores a pair of int values, whereas `PointF` stores a pair of float values. In addition to these values, these structures also provide a set of static methods and operators to perform basic operation on points.

Table 1.8 summarizes all the structures defined in the `System.Drawing` namespace.

TABLE 1.8

`System.Drawing` Namespace STRUCTURES

Structure	*Description*
`CharacterRange`	Specifies a range of character positions within a string.
`Color`	Specifies a color structure that has 140 static properties, each representing the name of a color. In addition, it also has four properties—`A`, `R`, `G`, and `B`—which specify the value for the `Alpha` (level of transparency), `Red`, `Green`, and `Blue` portions of the color. A `Color` value can be created by using any of its three static methods: `FromArgb()`, `FromKnownColor()`, and `FromName()`.
`Point`	Stores an ordered pair of integers, x and y, that defines a point in a two-dimensional plane. You can create a `Point` value by using its constructor. The `Point` structure also provides a set of methods and operators for working with points.

Structure	*Description*
`PointF`	Specifies a `float` version of the `Point` structure.
`Rectangle`	Stores the location and size of a rectangular region. You can create a `Rectangle` structure by using a `Point` structure and a `Size` structure. `Point` represents the top-left corner, and `Size` specifies the width and height from the given point.
`RectangleF`	Specifies a `float` version of the `Rectangle` structure.
`Size`	Represents the size of a rectangular region with an ordered pair of width and height.
`SizeF`	Specifies a `float` version of the `Size` structure.

Drawing Text on a Form

The `Graphics` class provides a `DrawString()` method that can be invoked on a Graphics object to render a text string on the drawing surface. There are six different forms in which the `DrawString()` method can be used. In this section, I discuss three of them; the other three forms are the same, but with one extra argument of `StringFormat` type that specifies alignment and line spacing information.

STEP BY STEP

1.13 Drawing Text on a Form

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_13` and click the Open button.

2. Open the Properties window. Search for the `Paint` event of the form, and double-click the row that contains the `Paint` event. Modify its event handler to look like this:

```
private void StepByStep1_13_Paint(object sender,
     System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
    String str = String.Format(
```

continues

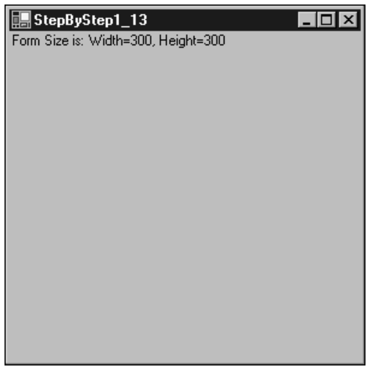
continued

```
         "Form Size is: Width={0}, Height={1}",
         Width, Height);
      grfx.DrawString(str, Font, Brushes.Black, 0, 0);
}
```

3. Add the following code for the `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_13());
}
```

4. Set the form as the startup object. Run the project. The form displays a string of text in black, showing the width and height of the form. Figure 1.20 shows the result.



Form Size is: Width=300, Height=300

FIGURE 1.20
You can draw text on a Windows form by using the `DrawString` method of the `Graphics` class.

In Step by Step 1.13, why did I choose to write the code within an event handler? There are two reasons, the foremost being that the `Paint` event handler provides access to the `Graphics` object. Second, the `Paint` event is fired whenever the form is redrawn, which includes when the form is first shown and when the form is restored from its minimized state, as well as when the form is shown after a window overlapping it is removed. Therefore, a `Paint` event handler is an appropriate place to put the code that you want to be executed whenever a form is redrawn.

The code gets the `Graphics` object through the `Graphics` property of the `PaintEventArgs` argument of the `Paint` event handler. The next step is to call the `DrawString()` method to draw text on the form. The `DrawString()` method used in Step by Step 1.13 takes five arguments and has the following signature:

```
public void DrawString(
   string, Font, Brush, float, float);
```

The first argument, string, is the string to be displayed. I use the `String.Format()` method to format the string.

The second argument, `Font`, is the font of the string. In Step by Step 1.13 I chose to display the string by using the default font of the current form through the `Font` property.

The third parameter, `Brush`, is the type of brush. The Brushes enumeration provides a variety of `Brush` objects, each with a distinct color. I chose the `Brushes.Black` value to draw text in black.

The fourth and fifth properties, `float` and `float`, specify the x and y locations for the point that marks the start of the string on the form. Both of these values are required to be of `float` type. The `0` value that the code contains is implicitly converted to a `float` value.

You might have noticed that when you resize the form in Step by Step 1.13, the `Paint` event is not triggered. An obvious idea for improving this form is to have it dynamically reflect the size of the form as you resize it. What event should you handle to do that? The `Resize` event. Step by Step 1.14 explains how to do this.

EXAM TIP

Unicode Support and `DrawString`
GDI+ and hence the Windows Forms library have full support for Unicode. This means that you can draw text in any language supported by the operating system.

STEP BY STEP

1.14 Using the `Invalidate()` Method

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_14` and click on the Open button.
2. Follow steps 2 and 3 from Step by Step 1.13 to include the `Paint` event handler code and `Main()` method to run the form `StepByStep1_14`.
3. Open the Properties window of the form `StepByStep1_14`. Search for the `Resize` event, and double-click on the row that contains the `Resize` event. Modify its event handler so that it looks like this:

```
private void StepByStep1_14_Resize(
   object sender, System.EventArgs e)
{
    // Call the Invalidate method
    Invalidate();
}
```

4. Set the form as the startup object. Run the project and notice that the form constantly modifies the text as it is resized.

After you complete Step by Step 1.14, the program works as desired. What's the deal with the `Invalidate()` method? The `Invalidate()` method causes the paint message to be sent to the form.

EXAM TIP

Invalidate() Method Calls When you call the Invalidate() method without any parameters, the Paint event is called for the entire area. If only a particular portion of the control needs to be refreshed, then calling Invalidate() for the entire area is rather taxing on application performance. In such a case you should call Invalidate() with a Rectangle parameter that specifies the portion of the control that you are interested in refreshing.

As a result, the Paint event handler is called. So this handy method can be called whenever the code in the Paint event handler needs to be executed. In Step by Step 1.14, the code makes a call to the Invalidate() method whenever the form is resized. The Invalidate() method is available in various forms, and you can refresh a specific portion of a form by using one of these forms.

Given the frequent requirement of calling Paint whenever Resize is fired, the Windows Forms library designers created a useful property: ResizeRedraw. This is a protected property that the Form class inherits from the Control class. When ResizeRedraw is set to true, it instructs a control (or a form, in this case) to redraw itself when it is resized. Its default value is false. Step by Step 1.15 shows how to use ResizeRedraw.

STEP BY STEP

1.15 Using the ResizeRedraw Property

1. Open the project 316C01. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form StepByStep1_15 and click the Open button.

2. Follow steps 2 and 3 from Step by Step 1.13 to include the Paint event handler code and Main() method to run the form StepByStep1_15.

3. Switch to the code view. Modify the constructor of the form so that the modified version looks like this:

```
public StepByStep1_15()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    // Paint when resized
    this.ResizeRedraw = true;
}
```

4. Run the project and notice that the form paints its drawing surface whenever you resize the form (just as it does in Step by Step 1.14, although the implementation is different).

The form's constructor is a good place to set the `ResizeRedraw` property. You could alternatively write it inside the `Main()` method itself.

As one more enhancement to Step by Step 1.15, you can center the text programmatically within the form. To do so, first you need to find the coordinates of the center of the form. You can find the horizontal distance by dividing the width of the client area (`ClientSize.Width`) by 2, and you can find the vertical distance by dividing the height of the client area (`ClientSize.Height`) by 2. The `ClientSize` properties give you access to a `Size` structure that represents the size of the client area of the form. But this does not really center the text onscreen because it simply causes the text to start from the center, and depending on how long it is, it might appear toward the right of the center. You need to adjust the coordinates of the center point according to the size of the string. Keep in mind that the size of the string can vary depending on what font you use for the text. A safe way to determine string size is to use the `MeasureString()` method of the `Graphics` object, as shown in the following code segment:

```
SizeF stringSize = grfx.MeasureString(str, Font);
```

You can then calculate the modified coordinates for placing the string as `x` = (`ClientSize.Width—stringSize.Width`)/2 and `y` = (`ClientSize.Height—stringSize.Height`)/2.

There is an alternative approach, however. You can use the `StringFormat` object in the following signature of the `DrawString()` method:

```
public void DrawString(
   string, Font, Brush, PointF, StringFormat);
```

The `StringFormat` argument lets you specify the text alignment and spacing options for the text. Step by Step 1.16 uses this form of `DrawString()` to center text onscreen.

STEP BY STEP

1.16 Drawing Text on a Form

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_16` and click the Open button.

continues

continued

2. Open the Properties window of the form. Search for the `Paint` event, and double-click the row that contains the `Paint` event. Modify its event handler so that it looks like this:

```
private void StepByStep1_16_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
    String strText = String.Format(
        "Form Size is: Width={0}, Height={1}",
        Width, Height);
    PointF pt = new PointF(ClientSize.Width/2,
        ClientSize.Height/2);

    // Set the horizontal and vertical alignment
    //using StringFormat object
    StringFormat strFormat = new StringFormat();
    strFormat.Alignment = StringAlignment.Center;
    strFormat.LineAlignment = StringAlignment.Center;

    // Create Font and Brush objects
    Font fntArial = new Font("Arial", 12);
    Brush brushColor = new SolidBrush(this.ForeColor);
    // Call the DrawString method
    grfx.DrawString(
       strText, fntArial, brushColor, pt, strFormat);
}
```

3. Modify the constructor of the form so that the modified version looks like this:

```
public StepByStep1_16()
{
    // Default Code in the constructor
    // Paint when resized
    this.ResizeRedraw = true;
}
```

4. Add the following code for the `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_16());
}
```

5. Set the form as the startup object. Run the project and resize the form. Notice that the text is displayed in the center of the form.

The code begins by calculating the center coordinates of the form. When you have the coordinates of the center, you want the string to be horizontally centered at that point. Calling `StringAlignment.Center` does this job. `StringAlignment` is an enumeration that is available in the `System.Drawing` namespace that specifies the location of the alignment of the text.

By default, when the `DrawString()` method draws a string, it aligns the top of the string with its x-coordinate value. This does not make much difference if the height of the text itself is not great, but as you increase the size of the font, text begins to hang down, starting from the x-axis. To center the text vertically within its own line, you can set the `LineAlignment` property of the `StringFormat` object to `StringAlignment.Center`.

Note the use of `Font` and `Brush` objects in Step by Step 1.16. The code creates a new `Font` object and specifies a font name and size. I recommend that you change its value and experiment with it. Rather than use the brush specified by the `Brushes.Black` value, the code in Step by Step 1.16 creates a Brush object that takes the value of its color from the current form's `ForeColor` property. If you change the `ForeColor` property of the form by using the Properties window, the change is automatically reflected here. Using a brush based on the `ForeColor` property of the form is a good idea as compared to using an absolute value such as `Brushes.Black` for a brush. For example, if the form designer has set the `BackColor` property of the form to `black` and the `ForeColor` property to `white`, text drawn using `Brushes.Black` would not be visible, but the brush made from the `ForeColor` property would be visible.

Drawing Shapes

The `Graphics` class allows you to draw various graphical shapes such as arcs, curves, pies, ellipses, rectangles, images, paths, and polygons. Table 1.9 lists some important drawing methods of the `Graphics` class.

TABLE 1.9

SOME IMPORTANT DRAWING METHODS OF THE `Graphics` CLASS

Method	*Description*
`DrawArc()`	Draws an arc that represents a portion of an ellipse
`DrawBezier()`	Draws a Bézier curve defined by four points
`DrawBeziers()`	Draws a series of Bézier curves
`DrawClosedCurve()`	Draws a closed curve defined by an array of points
`DrawCurve()`	Draws a curve defined by an array of points
`DrawEllipse()`	Draws an ellipse defined by a bounding rectangle specified by a pair of coordinates, a height, and a width
`DrawIcon()`	Draws the image represented by the specified `Icon` object at the given coordinates
`DrawImage()`	Draws an `Image` object at the specified location, preserving its original size
`DrawLine()`	Draws a line that connects the two points
`DrawLines()`	Draws a series of line segments that connect an array of points
`DrawPath()`	Draws a `GraphicsPath` object
`DrawPie()`	Draws a pie shape defined by an ellipse and two radial lines
`DrawPolygon()`	Draws a polygon defined by an array of points
`DrawRectangle()`	Draws a rectangle specified by a point, a width, and a height
`DrawRectangles()`	Draws a series of rectangles
`DrawString()`	Draws the given text string at the specified location, with the specified `Brush` and `Font` objects

In Step by Step 1.17 you use some of the methods shown in Table 1.9 to draw shapes on a form's surface.

STEP BY STEP

1.17 Using the `Draw` Methods of the `Graphics` Class

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_17` and click the Open button.
2. Open the Properties window. Search for the `Paint` event, and double-click the row that contains the `Paint` event. You are taken to the code view.
3. On the top of the code view, along with the list of other `using` directives, add the following line of code:

```
using System.Drawing.Drawing2D;
```

4. Modify the code in the `Paint` event handler to look like this:

```
private void StepByStep1_17_Paint(object sender,
   System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
          // Set the Smoothing mode
          // to SmoothingMode.AntiAlias
    grfx.SmoothingMode = SmoothingMode.AntiAlias;
          // Create Pen objects
    Pen penYellow = new Pen(Color.Blue, 20);
    Pen penRed = new Pen(Color.Red, 10);
    // Call Draw methods
    grfx.DrawLine(Pens.Black, 20, 130, 250, 130);
    grfx.DrawEllipse(penYellow, 20, 10, 100, 100);
    grfx.DrawRectangle(penRed, 150, 10, 100, 100);
}
```

5. Add the following code to insert the `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_17());
}
```

6. Set the form as the startup object. Run the project, and you see the form as displayed in Figure 1.21.

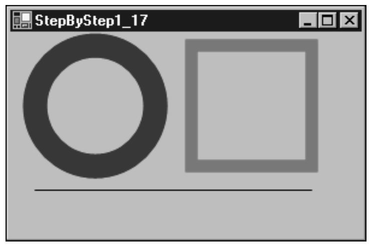


FIGURE 1.21
You can call `Draw` methods of the `Graphics` class to draw various graphic shapes.

The `Draw` methods used in Step by Step 1.17 take five arguments:

- The first argument of each method is the `Pen` object that is used to draw the shape. There are several ways you can create a `Pen` object. The simplest way is to use a readymade `Pen` object from the `Pens` class. Or you can create a Pen object by using the `Pen` class constructor. Table 1.10 lists the different pen-related classes that are available in the `System.Drawing` namespace.
- The second and third arguments are the x- and y-coordinates of the upper-left corner where the desired shape is to be drawn.
- The fourth and fifth parameters indicate the width and height of the desired shape to be drawn. In the case of `DrawLine`, these indicate the x- and y-coordinates of the ending point of the line drawn.

TABLE 1.10

PEN-RELATED CLASSES IN THE `System.Drawing` NAMESPACE

Class	*Description*
`Pen`	Defines an object used to draw lines and curves.
`Pens`	Provides 140 static properties, each representing a pen of a color supported by Windows Forms.
`SystemPens`	Provides a set of static properties, each named after a Windows display element such as `ActiveCaption`, `WindowText`, and so on. Each of these properties returns a `Pen` object with a width of 1.

The reason you must include a reference to the `System.Drawing.Drawing2D` namespace in the program shown in Step by Step 1.17 is that you are using an enumeration named `SmoothingMode`. This enumeration class is defined in the namespace `System.Drawing.Drawing2D`, so a reference to the namespace must be present in the program if the C# compiler is to uniquely identify it.

The `Graphics` object has a property named `SmoothingMode` that can take the values of the `SmoothingMode` enumeration type. Table 1.11 summarizes these values. The `SmoothingMode` property specifies the quality of rendering. The Windows Forms library supports antialiasing, which produces text and graphics that appear to be smooth.

NOTE

Antialiasing *Antialiasing* is a technique for rendering images where partially transparent pixels are drawn close to the opaque pixels present at the edges of a drawing. This actually makes the edges kind of fuzzy, but this effect makes the edges appear smoother to human eyes than the original form. Because there are extra efforts involved in antialiasing, it makes rendering of graphics slower than not using antialiasing.

TABLE 1.11

`SmoothingMode` ENUMERATION MEMBERS

Member Name	*Description*
`AntiAlias`	Specifies an antialiased rendering.
`Default`	Specifies no antialiasing. Same as `None`.
`HighQuality`	Specifies a high-quality, low-performance rendering. Same as `AntiAlias`.
`HighSpeed`	Specifies a high-performance, low-quality rendering. Same as `None`.
`Invalid`	Specifies an invalid mode, raises an exception.
`None`	Specifies no antialiasing.

In addition to the `Draw` methods, the `Graphics` class also provides a variety of `Fill` methods (see Table 1.12). You can use these methods to draw a solid shape on a form.

TABLE 1.12

`Fill` METHODS OF THE `Graphics` CLASS

Method Name	*Description*
`FillClosedCurve()`	Fills the interior of a closed curve defined by an array of points
`FillEllipse()`	Fills the interior of an ellipse defined by a bounding rectangle
`FillPath()`	Fills the interior of a `GraphicsPath` object
`FillPie()`	Fills the interior of a pie section defined by an ellipse and two radial lines
`FillPolygon()`	Fills the interior of a polygon defined by an array of points

continues

TABLE 1.12 *continued*

FILL METHODS OF THE GRAPHICS CLASS

Method Name	*Description*
`FillRectangle()`	Fills the interior of a rectangle specified by a point, a width, and a height
`FillRectangles()`	Fills the interiors of a series of rectangles
`FillRegion()`	Fills the interior of a `Region` object

STEP BY STEP

1.18 Using the `Fill` Methods of the `Graphics` Class

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_18` and click the Open button.
2. Open the Properties window of the form. Search for the `Paint` event, and double-click the row that contains the `Paint` event. You are taken to the code view.
3. Modify the code in the `Paint` event handler to look like this:

```
private void StepByStep1_18_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
    // Create Brush objects
    Brush brushRed = new SolidBrush(Color.Red);
    Brush brushYellow = new SolidBrush(
        Color.FromArgb(200, Color.Yellow));
    // Call Fill methods
    grfx.FillEllipse(brushRed, 20, 10, 80, 100);
    grfx.FillRectangle(brushYellow, 60, 50, 100, 100);
}
```

4. Add the following code to insert the `Main()` method:

```
[STAThread]
static void Main()
{
   Application.Run(new StepByStep1_18());
}
```

5. Set the form as the startup object. Run the project. An overlapping red ellipse and yellow rectangle appear on the form, as shown in Figure 1.22.

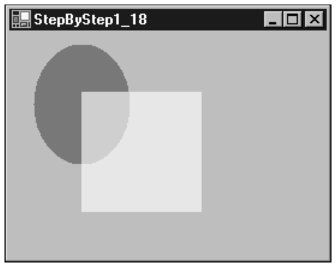


FIGURE 1.22
You can use `Fill` methods of the `Graphics` class to draw solid shapes.

The syntax of the `Fill` methods is somewhat similar to that of corresponding `Draw` methods. The only difference is that the `Fill` methods use a `Brush` object to fill a drawing object with a color.

Creating the yellow brush in Step by Step 1.18 looks interesting. While creating this color, you do an "alpha-blending" with the yellow color, to get a kind of transparent yellow color that is used to produce an overlay effect.

You use a `SolidBrush` object in Step by Step 1.18 to fill shapes. Other types of brushes can be used to create fancy filling effects; Table 1.13 lists them.

TABLE 1.13

TYPES OF BRUSHES IN THE `System.Drawing` AND `System.Drawing.Drawing2D` NAMESPACES

Class	*Description*
`Brush`	Is an abstract base class that is used to create brushes such as `SolidBrush`, `TextureBrush`, and `LinearGradientBrush`. These brushes are used to fill the interiors of graphical shapes such as rectangles, ellipses, pies, polygons, and paths.
`Brushes`	Provides 140 static properties, one for the name of each color supported by Windows forms.
`HatchBrush`	Allows you to fill the region by using one pattern from a large number of patterns available in the `HatchStyle` enumeration.
`LinearGradientBrush`	Is used to create two-color gradients and multicolor gradients. By default the gradient is a linear gradient that moves from one color to another color along the specified line.

continues

TABLE 1.13 *continued*

TYPES OF BRUSHES IN THE System.Drawing AND System.Drawing.Drawing2D NAMESPACES

Class	*Description*
`SolidBrush`	Defines a brush of a single color. Brushes are used to fill graphics shapes, such as rectangles, ellipses, pies, polygons, and paths.
`SystemBrushes`	Provides a set of static properties, each named after a Windows display element, such as `ActiveCaption`, `WindowText`, and so on. Each of these properties returns a `SolidBrush` object that represents the color for its matching Windows display element.
`TextureBrush`	A class that contains properties for the `Brush` object that use images to fill the interior of shapes.

STEP BY STEP

1.19 Using Different Brush Types

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_19` and click the Open button.
2. Open the Properties window. Search for the `Paint` event, and double-click the row that contains the `Paint` event. You are taken to the code view.
3. On the top of the code view, along with the list of other `using` directives, add the following lines of code:

```
using System.Drawing.Drawing2D;
```

4. Modify the code in the `Paint` event handler so that it looks like this:

```
private void StepByStep1_19_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
    // Create a HatchBrush object
    // Call FillEllipse method by passing
    // the created HatchBrush object
    HatchBrush hb = new HatchBrush(
        HatchStyle.HorizontalBrick,
       Color.Blue, Color.FromArgb(100, Color.Yellow));
    grfx.FillEllipse(hb, 20, 10, 100, 100);

    // Create a TextureBrush object
    // Call FillEllipse method by passing the
    // created TextureBrush object
    Image img = new Bitmap("sunset.jpg");
    Brush tb = new TextureBrush(img);
    grfx.FillEllipse(tb, 150, 10, 100, 100);

    // Create a LinearGradientBrush object
    // Call FillEllipse method by passing
    // the created LinearGradientBrush object
    LinearGradientBrush lb = new LinearGradientBrush(
        new Rectangle(80, 150, 100, 100),
        Color.Red, Color.Yellow,
           LinearGradientMode.BackwardDiagonal);
    grfx.FillEllipse(lb, 80, 150, 100, 100);
}
```

5. Add the following code to insert the `Main()` method:

```
[STAThread]
static void Main()
{
    Application.Run(new StepByStep1_19());
}
```

6. Set the form as the startup object. Run the project. Notice that ellipses filled in various styles are displayed in the form, as shown in Figure 1.23.

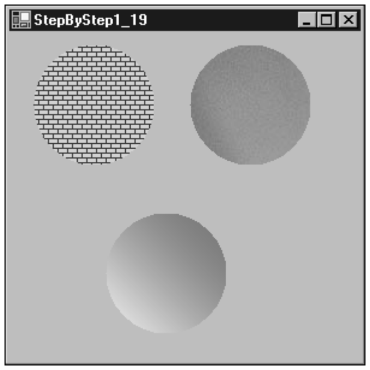


FIGURE 1.23
You can create fancy objects by using different brush types.

In Step by Step 1.19 you use three different `Brush` objects. The `TextureBrush` class is part of the `System.Drawing` namespace, and the other two classes (`HatchBrush` and `LinearGradientBrush`) are members of the `System.Drawing.Drawing2D` namespace.

Using `HatchBrush`, you filled the ellipse with the `HatchStyle` named `HorizontalBrick`. The `TextureBrush` class uses an image to fill the interior of a shape, and in Step by Step 1.19, the image is assumed to be in the same directory as the `.exe` file. If you have the image in some other directory, you need to change the path in the `Bitmap` constructor. The Bitmap name is a bit misleading, as it is actually capable of creating images from a variety of image formats, including BMP, GIF, JPG, and PNG.

In Step by Step 1.19 the code sets the gradient direction from the upper-right corner to the lower-left corner of the rectangle encapsulating the ellipse. Table 1.14 lists the enumeration values for `LinearGradientMode`.

TABLE 1.14

`LinearGradientMode` ENUMERATION VALUES

Member Name	*Description*
`BackwardDiagonal`	Specifies a gradient from upper-right to lower-left
`ForwardDiagonal`	Specifies a gradient from upper-left to lower-right
`Horizontal`	Specifies a gradient from left to right
`Vertical`	Specifies a gradient from top to bottom

Working with Images

The `System.Drawing.Image` class provides the basic functionality for working with images. However, the `Image` class is abstract, which means you can create an instance of it in your class. Instead of using the `Image` class directly, you can use the following classes that implement the `Image` class functionality:

- **`Bitmap`** This class is used to work with graphic files that store information in pixel-based data such as BMP, GIF, and JPEG formats.
- **`Icon`** This class creates a small bitmap that represents an Windows icon.
- **`MetaFile`** This class contains embedded bitmaps and/or sequences of binary records that represent a graphical operation such as drawing a line.

Step by Step 1.20 shows how to do basic operations with the `Bitmap` class.

STEP BY STEP

1.20 Creating and Rendering Images

1. Open the project `316C01`. In the Solution Explorer right-click the project name and select Add Windows Form from the context menu. Name the new form `StepByStep1_20` and click the Open button.

2. Open the Properties window. Search for the `Paint` event, and double-click the row that contains the `Paint` event. You are taken to the code view.

3. On the top of the code view, along with the list of other `using` directives, add the following lines of code:

```
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
```

4. Modify the code in the `Paint` event handler to look like this:

```
private void StepByStep1_20_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx = e.Graphics;
    grfx.DrawImage(new Bitmap("SampleImage.png"),
        ClientRectangle);
}
```

5. Add the following code after the event handling code from step 4:

```
[STAThread]
static void Main()
{
    // Create a Bitmap object
    Bitmap bmp = new Bitmap(
        800,600,PixelFormat.Format32bppArgb);
    // Create a Graphics object using FromImage method
    Graphics grfx = Graphics.FromImage(bmp);
    // Call the Fill Rectangle method
    // to create an outer rectangle
    grfx.FillRectangle(new SolidBrush(Color.White),
        new Rectangle(0,0,800,600));
    // Create Font and RectangleF object
    Font fntText = new Font("Verdana", 20);
    RectangleF rect = new RectangleF(
       100, 100, 250, 300);
```

continues

continued

```
    // Fill the InnerRectangle
    grfx.FillRectangle(
        new SolidBrush(Color.AliceBlue), rect);
    // Add the text to the Inner Rectangle
    grfx.DrawString("Sample Text", fntText,
      new SolidBrush(Color.Blue), rect);
    // Draw a closed curve
    Pen penBlack = new Pen(Color.Black, 20);
    penBlack.DashStyle = DashStyle.Dash;
    penBlack.StartCap = LineCap.Round;
    penBlack.EndCap = LineCap.Round;
    grfx.DrawClosedCurve(penBlack, new Point[] {
                         new Point(50, 50),
                         new Point(400, 50),
                         new Point(400, 400),
                         new Point(50, 400)});
    // Save the newly created image file
    bmp.Save("SampleImage.png", ImageFormat.Png);
    Application.Run(new StepByStep1_20());
}
```

6. Set the form as the startup object. Run the project. The code should create an image and render it in the form, as shown in Figure 1.24.

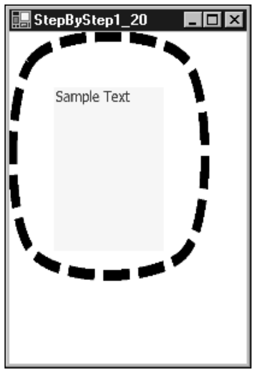


FIGURE 1.24
You can use the `Bitmap` class to work with a variety of image formats.

In Step by Step 1.20 you first create an image and then render it on the form's surface. The image creation code is written in the `Main()` method, which means it is executed before the application creates the `Form` object. Three key steps are related to image manipulation:

1. You need to create a `Bitmap` object that can be used to work on images. The code for this creates the object by specifying its size in pixels and also specifying a format value from the `PixelFormat` enumeration, which belongs to the `System.Drawing.Imaging` namespace. The value `Format32bppArgb` specifies that there are 32 bits of data associated with each pixel in the image. Out of these 32 bits, 8 bits each are used for the alpha, red, green, and blue components of the pixel.

2. You need to get a `Graphics` object from the `Bitmap` object. This `Graphics` object can be used to draw on the surface of the drawing.

3. You need to call the `Save()` method on the `Bitmap` object. This method supports a variety of formats for saving graphics; these formats are available as static public properties of the `ImageFormat` class. The `ImageFormat` class is a member of the `System.Drawing.Imaging` namespace. Some of the possible properties are `Bmp`, `Gif`, `Icon`, `Jpeg`, `Png`, `Tiff`, and `Wmf`.

The rest of the code in Step by Step 1.20 draws a piece of text and a curved line on the image before saving it. Note the various properties of the `Pen` object that are used in this program. The code uses the customized `Pen` object to create a boundary around the image. Table 1.15 summarizes some important properties of the `Pen` class.

TABLE 1.15

SOME IMPORTANT PROPERTIES OF THE `Pen` CLASS

Property	*Description*
`Alignment`	Specifies the alignment for the Pen object
`Brush`	Specifies a Brush object that determines the attributes of the Pen object
`Color`	Specifies the color of the Pen object
`DashCap`	Specifies the cap style used at the end of the dashes that make up dashed lines
`DashPattern`	Specifies the array of custom dashes and spaces
`DashStyle`	Specifies the style used for dashed lines
`EndCap`	Specifies the cap style used at the end of lines
`LineJoin`	Specifies the join style for the ends of two consecutive lines
`PenType`	Specifies the style of lines
`StartCap`	Specifies the cap style used at the beginnings of lines
`Width`	Specifies the width of the `Pen` object

GUIDED PRACTICE EXERCISE 1.3

You are a Windows developer for SpiderWare, Inc. The Windows application you are working on should have a form that allows users to create new designs for Spider-Webs. The requirement itself is simple: The form must have a white background, where users can design using a thin black pen. Users often make mistakes while they are designing, so there should be a mechanism to erase part of the design. Users say that they will be comfortable using the left mouse button for drawing and the right mouse button for erasing. You also noted that users of the application all have high-end machines, and they want sharper-looking designs.

How would you design such a form?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Open the project `316C01`. Add a Windows form with the name `GuidedPracticeExercise1_3` to this project.
2. Open the Properties window for the form. Change the form's `BackColor` property to `White`.
3. Look for the event named `MouseMove` and double-click it. This inserts a `MouseMove` event handler for you and switches you to the code view. In the code view, add the following statement at the top of the program:

   ```
   using System.Drawing.Drawing2D;
   ```

4. Before the `MouseMove` event handler code add the following line:

   ```
   Point ptPrevPosition = new Point(-1, -1);
   ```

5. Modify the `MouseMove` event handler to look like this:

   ```
   private void GuidedPracticeExercise1_3_MouseMove(
     object sender, System.Windows.Forms.MouseEventArgs e)
   {
       if(ptPrevPosition.X == -1)
       {
   ```

```
            // Set the previous position x-y co-ordinate
            // to the current x-y co-ordinate
            ptPrevPosition = new Point(e.X, e.Y);
        }
        Point ptCurrPosition = new Point(e.X,e.Y);
        // Get the Graphics object by calling
        // Graphics.FromHwnd() method
        Graphics g = Graphics.FromHwnd(this.Handle);
        g.SmoothingMode = SmoothingMode.AntiAlias;
        // Check if the left mouse button is pressed
        if(e.Button == MouseButtons.Left)
        {
            // Draw a line from the previous position to
            // current position using Black color
            g.DrawLine(new Pen(Color.Black),
                ptPrevPosition, ptCurrPosition);
        }
        // Check whether right mouse button is pressed
        else if(e.Button == MouseButtons.Right)
        {
            // Draw a line from the previous position to
            // current position using Form's BackColor
            g.DrawLine(new Pen(this.BackColor, 4),
               ptPrevPosition, ptCurrPosition);
        }
        // Set the Previous position to current position
        ptPrevPosition = ptCurrPosition;
    }
```

6. Insert the following `Main()` method just after the event handling code:

```
[STAThread]
static void Main()
{
    Application.Run(new GuidedPracticeExercise1_3());
}
```

7. Set the form as the startup object and execute the program.

If you have difficulty following this exercise, review the sections "Event Handling" and "Drawing Shapes." The text and examples presented in those sections should help you relearn this material. After doing that review, try this exercise again.

REVIEW BREAK

- Windows forms follow a two-dimensional coordinate system. A point is an addressable location in this coordinate system.
- The `Graphics` object gives you access to a drawing surface. You can use it to draw lines, text, curves, and a variety of shapes.
- The `ResizeRedraw` property, when set to `true`, instructs the form to redraw itself when it is resized. It's a good programming practice to design forms that resize their contents based on their size. The Resize event of a form can also be used to program the resizing logic.
- The `Graphics` class provides a set of `Draw` methods that can be used to draw shapes such as rectangles, ellipses, and curves on a drawing surface. The `Graphics` class also provides a set of `Fill` methods that can be used to create solid shapes.
- An object of the `Bitmap` class gives you access to image manipulation. The `System.Drawing` namespace classes can understand a variety of image formats.

CHAPTER SUMMARY

The .NET Framework is a standards-based multilanguage platform for developing next-generation applications. Visual Studio .NET provides a productive IDE for developing .NET Framework applications.

The .NET FCLs include classes for developing Windows-based desktop and distributed applications. Visual Studio .NET provides the Windows Forms Designer, which allows you to visually drag and drop components and create applications. Various Step by Step exercises in this chapter help you familiarize yourself with the development environment and its key concepts.

A Windows form is the place where you assemble the user interface of an application. Form is a class that provides various properties through which you can get or set a form's characteristics. In this chapter you have learned how to manipulate a form's properties and how to add a custom property to a form. You have also learned how to derive from an existing form and extend the functionality of an existing form by adding your own properties and methods.

Event handling plays a key role in user interface–based programming; through event handling, you respond to various events that are fired as a result of user actions and that make programs interactive. This chapter discusses various ways to handle events. In Chapter 4 you will learn how to define your own events.

In this chapter you have also learned how to use the classes from the .NET Framework that implement graphics functionality. You have seen how to draw text, lines, and shapes, and you have learned how to work with other key graphics elements such as brushes, colors, and pens.

Chapter 2 talks more about various user interface elements that allow rapid development of powerful and interactive Windows applications.

KEY TERMS

- application
- attribute
- class
- constructor
- delegate
- enumeration
- event
- event handling
- FCL
- field
- garbage collection
- GDI
- inheritance
- IL
- JIT compilation
- managed code
- namespace
- .NET Framework
- property
- structure
- visual inheritance
- Windows Forms Designer

APPLY YOUR KNOWLEDGE

Exercises

1.1 Responding to Keyboard Input

The Windows forms libraries provide controls such as `TextBox` and `RichTextBox` that you can use to process keyboard input from users. But sometimes you might want to program the keyboard yourself. In this exercise you will learn how to capture keyboard events and respond to them. You will design a very basic text editor named NEN (Not Even Notepad) that will let you type on a form surface and also let you edit the text by using the Backspace key.

Estimated time: 20 minutes

1. Create a new Visual C# .NET Windows application in the Visual Studio .NET IDE.

2. Add a new form to the Visual C# .NET project. Change the `Text` property of the form to `Not Even Notepad` and change the `BackColor` property to `White`.

3. In the code view, add the following `using` directive at the top:

```
using System.Text;
```

4. Declare a private variable just before the form's constructor code:

```
private StringBuilder text;
```

5. In the constructor, initialize the private variable from step 4 by including the following line of code:

```
this.text = new StringBuilder();
```

6. Add the following code in the `KeyPress` event handler of the form:

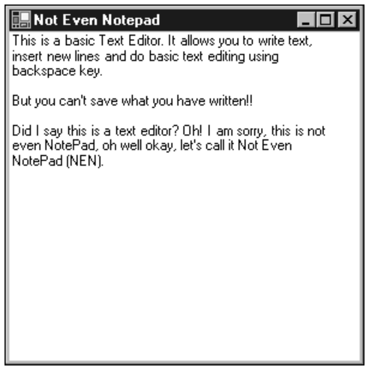
```
private void Exercise1_1_KeyPress(
  object sender,
  System.Windows.Forms.KeyPressEventArgs e)
{
    // Check which key is pressed
    switch(e.KeyChar)
    {
        case '\b':
            // Backspace key is pressed
            if (sbText.Length > 0)
                sbText.Remove(
                  sbText.Length-1, 1);
            break;
        case '\r':
        case '\n':
            // Enter key is pressed
            sbText.Append('\n');
            break;
        default:
            // Other keys are pressed
            sbText.Append(e.KeyChar);
            break;
    }
    // Paint the form
    Invalidate();
}
```

7. In the form's `Paint` event handler, add the following code:

```
private void Exercise1_1_Paint(
  object sender,
  System.Windows.Forms.PaintEventArgs e)
{
    Graphics grfx =
      ((Form) sender).CreateGraphics();
    grfx.DrawString(text.ToString(), Font,
       Brushes.Black,ClientRectangle );
}
```

8. Insert the `Main()` method and set the form as the startup object of the project. Execute the application. You should see a form onscreen that shows no blinking cursor on it, but when you start typing, the form should show text. You can press the Enter key to start a new paragraph and press the Backspace key to make any changes (see Figure 1.25).

APPLY YOUR KNOWLEDGE



This is a basic Text Editor. It allows you to write text, insert new lines and do basic text editing using backspace key.

But you can't save what you have written!!

Did I say this is a text editor? Oh! I am sorry, this is not even NotePad, oh well okay, let's call it Not Even NotePad (NEN).

FIGURE 1.25
The `KeyPress` event allows you to capture keystrokes.

The `KeyPress` event is fired when you press a key on the keyboard. In its event handler, the code uses these keypresses to modify a `StringBuilder` object that stores the text for the small text editor. Then it calls the `Invalidate()` method, which in turn generates a call to the `Paint` event handler that is actually drawing the text onscreen. So in fact every keypress results in a total repainting of the form. Chapter 2 talks about better ways of doing this task.

1.2 Getting a List of Installed Fonts

Several Windows applications allow you to change the font of displayed text. They normally give you a list of fonts installed on your system to choose from. How do they get this list? This exercise shows you how you can work with font-related classes in the `System.Drawing` namespace to display a list of installed fonts.

Estimated time: 15 minutes

1. Add a new form to your Visual C# .NET project.
2. Change the `Text` property of the form to `List of Installed Fonts` and change the `BackColor` property to `White`.
3. Add a `Paint` event handler to the form, and add the following code to it:

```
private void Exercise1_2_Paint(
   object sender,
   System.Windows.Forms.PaintEventArgs e)
{
    // Set the y coordinate to 0
    int intYCoord=0;

    // Create a Black color SolidBrush
    SolidBrush brush =
       new SolidBrush(Color.Black);

    // Iterate through the
    // FontFamily.Families
    for(int intI=0;intI <
      FontFamily.Families.Length; intI++)
    {
        FontStyle fontStyle =
          FontStyle.Regular;

        // Check whether Regular style
        // is available
        if (!FontFamily.Families[
           intI].IsStyleAvailable(
           FontStyle.Regular))
            fontStyle = FontStyle.Italic;

        // Check whether Italic style
        // is available
        if (!FontFamily.Families[
           intI].IsStyleAvailable(
           FontStyle.Italic))
            fontStyle = FontStyle.Bold;
        // Create a Font object and
        // Draw the Font Name
        Font font  = new Font(
          FontFamily.Families[intI].Name,
          12, fontStyle);
        string strFontName =
            FontFamily.Families[intI].Name;
        e.Graphics.DrawString(strFontName,
           font, brush, 0, intYCoord );
        // Increase the Y Coordinate
        // with the Font Height
        intYCoord += font.Height;
    }

}
```

APPLY YOUR KNOWLEDGE

4. Insert the `Main()` method and set the form as the startup object of the project. Execute the application. You should see a form that displays a list of fonts, with each font name displayed in its own font. When you increase the height of the form, you see more lines that list the fonts (see Figure 1.26).

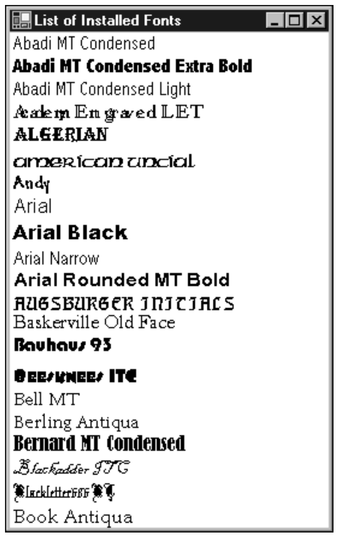

List of Installed Fonts
Abadi MT Condensed
Abadi MT Condensed Extra Bold
Abadi MT Condensed Light
Academy Engraved LET
ALGERIAN
american uncial
Andy
Arial
Arial Black
Arial Narrow
Arial Rounded MT Bold
AUGSBURGER INITIALS
Baskerville Old Face
Bauhaus 93
BEESKNEES ITC
Bell MT
Berling Antiqua
Bernard MT Condensed
Blackadder ITC
BlackletterRR
Book Antiqua

FIGURE 1.26
You can use the `FontFamily.Families` property to get all the `FontFamily` objects associated with the current `Graphics` context.

The `FontFamily.Families` property stores an array of `FontFamily` objects. The code iterates over this array to display each font name in its own font style. If you remove the two `if` statements that are used in this program, you get a runtime error because not all fonts support all font styles.

In Figure 1.26 you cannot see all the fonts because the list of fonts usually contains more fonts than the number of lines that can be displayed, even on a maximized form. You'll learn how to make the contents of a form scroll in Chapter 2.

1.3 Creating Nonrectangular Forms

All the forms that you have created in this chapter so far have been rectangular forms. The Windows Forms library also allows you to create nonrectangular forms. Nonrectangular forms can be used in various applications, such as games, device simulations, and multimedia applications. In this exercise you will see how to create a nonrectangular form.

Estimated time: 10 minutes

1. Add a new form to your Visual C# .NET project.
2. In the Properties window for the form, change the `FormBorderStyle` property to `None` and change the `BackColor` property to `FireBrick`.
3. Switch to the code view and include the following `using` directive at the top of the code:

```
using System.Drawing.Drawing2D;
```

4. In the constructor of the form, include the following lines of code:

```
public Exercise1_3()
{
   //
   // Default code of the constructor
   //
   // Create a GraphicsPath object
   // AddEllipse to the GraphicsPath object
   // Set the Form's Region property for
   // the Graphics Path region
   GraphicsPath gp = new GraphicsPath();
   gp.AddEllipse(25,25,250,250);
   this.Region = new Region(gp);
}
```

5. Insert the `Main()` method and set the form as the startup object of the project. Run the application. You should see a circular red form.

This code first defines an elliptical region, using the `GraphicsPath` object. The `GraphicsPath` object is then used to set the `Region` property of the form. The `Region` property instructs the operating system to hide any portion of the form that lies outside the elliptical region. As a result, the form is displayed as an ellipse.

Review Questions

1. Describe the difference between a public field and a public property.
2. What is the purpose of organizing classes in namespaces?
3. What property would you use to control the shape of the mouse pointer when it enters the client area of a Windows form?
4. How can you add a custom property to a form?
5. What is visual inheritance?
6. What are the two different approaches for event handling? What is the difference between them?
7. What is the `ResizeRedraw` property? When would you want to set it to `true` for a Windows form?
8. Describe at least two ways by which you can create a `Graphics` object for a Windows form.
9. What is the difference between a `Pen` object and a `Brush` object?

OBJECTIVES

This chapter covers the following Microsoft-specified objectives for the "Creating User Services" section of Exam 70-316, "Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET":

Add controls to a Windows form.

- **Set properties on controls.**
- **Load controls dynamically.**
- **Write code to handle control events and add the code to a control.**
- **Create menus and menu items.**

▶ Controls are the most visible part of a Windows application. The purpose of this objective is to test your knowledge of working with the most common Windows forms controls, including working with their properties, methods, and events.

Implement navigation for the user interface (UI).

- **Configure the order of tabs**

▶ When you place controls on a form, you need to provide a logical order of keyboard-based navigation for the controls. This exam objective covers how to achieve this logical order by using Visual Studio .NET.

CHAPTER 2

Controls

OUTLINE

STUDY STRATEGIES

- Experiment with the common Windows controls that are available in the Windows Forms Designer toolbox. Knowing their properties, methods, and events well will help you answer several exam questions. You will find several important members of various controls listed in tables throughout the chapter.
- Know how to handle events for Windows Forms controls. Make sure you read the section "Event Handling" in Chapter 1, "Introducing Windows Forms."
- Understand how to create controls dynamically. See Step by Step 2.2 and Step by Step 2.4 to get hands-on experience in loading controls dynamically.
- Know how to create menus and menu items. This includes creating both a main menu and context menus for an application.

INTRODUCTION

This chapter extends the concepts presented in Chapter 1, "Introducing Windows Forms," and discusses various aspects of user interface programming in more detail.

Windows forms controls are reusable components that encapsulate graphical user interface (GUI) functionality in Windows-based applications. The chapter starts by teaching you how to add various controls to a Windows form, how to set the properties of the controls, and how to program various events associated with the controls.

This discussion is followed by text that explains how to use common dialog boxes in applications and how to create custom dialog boxes for specific requirements.

This chapter also covers most of the commonly used controls that are available in the Visual Studio .NET toolbox. Controls are explained with examples that help you understand and appreciate how they function.

Next, the chapter teaches how to create a main menu and a context menu for Windows applications and how to associate menu items with specific actions.

ADDING CONTROLS TO A WINDOWS FORM

You can place controls on the surface of any container object. Container objects include Windows forms and panels and group box controls.

You can add controls to a form either programmatically or by using the Windows Forms Designer. Although the Windows Forms Designer provides an easy-to-use interface for adding controls to a form, you are likely to need to do some work in the code view to make programs more functional.

05 0789728230 CH02 11/21/02 1:20 PM Page 117

Adding Controls by Using the Windows Forms Designer

Add controls to a Windows form.

The Windows Forms Designer provides a toolbox that contains a variety of commonly used controls. You can drag and drop controls from the toolbox to a form and arrange them as required. You can activate the toolbox by selecting View, Toolbox or by pressing Ctrl+Alt+X. You see a rich set of controls on the Windows Forms tab of the Toolbox (see Figure 2.1). From the toolbox you can use different ways to add controls to a form or another container object, including the following:

- **Method 1**—You can select a control and draw it on the container surface by following these steps:

  1. Select a control by clicking on the control's icon in the toolbox (refer to Figure 2.1).
  2. Release the mouse button and move the mouse pointer to the position on the container where you want to draw the control.
  3. Hold down the mouse button and draw a rectangle on the container surface to indicate the size and position for the control instance.

- **Method 2**—You can drag a control onto the form at the desired location by following these steps:

  1. Select a form or another container control where you want to add a control.
  2. Drag the control's icon from the toolbox and drop it at the desired location on the container control. The control is added with its default size.

- **Method 3**—You can add a control to a form by double-clicking it, as in the following steps:

  1. Select a form or another container control where you want to add a control.

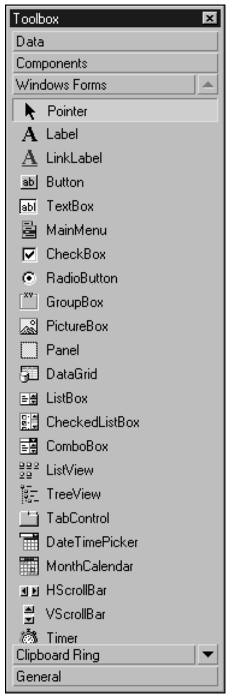


FIGURE 2.1
The Windows Forms Designer toolbox displays a variety of items for use in Visual Studio .NET projects.

2. Double-click the control's icon in the toolbox. This adds the control to the top-left corner of the form or other container control in its default size. You can now drag the control to its desired location on the container control.

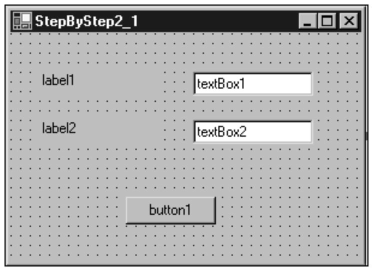


FIGURE 2.2
You can use the Windows Forms Designer to add controls to a form.

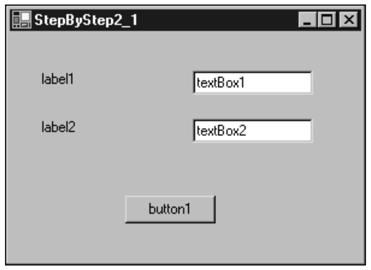


FIGURE 2.3
What you see at design time is what you get at runtime.

STEP BY STEP

2.1 Adding Controls to a Windows Form by Using the Windows Forms Designer

1. Create a new C# Windows application project in the Visual Studio .NET Integrated Development Environment (IDE). Name the project `316C02`.
2. Add a new Windows form to the project. Name it `StepByStep2_1`.
3. Select the toolbox. On the Windows Forms tab place two controls of type `Label` and `TextBox`, and then place a control of type `Button` on the form's surface. Arrange the controls as shown in Figure 2.2.
4. Insert a `Main()` method to launch the form and set the form as the startup object for the project.
5. Run the project. You should see a Windows form that looks like the form shown in Figure 2.3. Navigate among the controls by using the Tab key.

Adding Controls Programmatically

Add controls to a Windows form.

- **Load controls dynamically.**

It is possible to add controls to a form programmatically. When you do so, you must remember to follow these three steps:

1. Create a private variable to represent each of the controls you want to place on the form.

2. In the form, place code to instantiate each control and to customize each control, using its properties, methods, or events.
3. Add each control to the form's control collection.

Step by Step 2.2 demonstrates this process.

STEP BY STEP

2.2 Adding Controls to a Windows Form Programmatically

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_2`.

2. Switch to the code view and add the following variables just above the form's constructor code:

```
//create variables to hold controls
private Label lblName, lblPassword;
private TextBox txtName, txtPassword;
private Button btnLogin;
```

3. Add the following code to the form's constructor:

```
//specify the form's size
this.ClientSize = new System.Drawing.Size(272, 182);

//set up the label for prompting Name
lblName = new Label();
lblName.Text = "Name: ";
//Specify the location for proper placement
//the default location will be (0, 0) otherwise
lblName.Location = new Point(16, 16);

//set up label for prompting Password
lblPassword = new Label();
lblPassword.Text = "Password: ";
lblPassword.Location = new Point(16, 80);

//setup a text box that allows user to enter Name
txtName = new TextBox();
txtName.Location = new Point(152, 16);

//setup text box for entering password
txtPassword = new TextBox();
txtPassword.Location = new Point(152, 80);
txtPassword.PasswordChar = '*';
```

continues

continued

```
//set up a command button
btnLogin = new Button();
btnLogin.Text = "Login";
btnLogin.Location = new Point(96, 128);

//Add control to the form
//Method 1: Specify the current form as
//parent container for a control
lblName.Parent = this;

//Method 2: Add a control to form's control collection
this.Controls.Add(txtName);

//Method 3: Add an array of controls to
//form's control collection
this.Controls.AddRange(
  new Control[] {lblPassword, txtPassword, btnLogin});
```

4. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.

5. Run the project. The form is displayed, as shown in Figure 2.4.

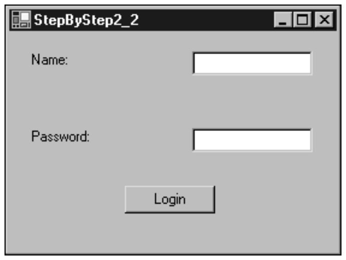


FIGURE 2.4
You can programmatically add controls to a Windows form.

When you create controls programmatically, more effort is involved in finding the exact location where you would like to display the controls on the form than in plotting the controls via the Windows Forms Designer. In exchange for this small inconvenience, the code view allows you to add more functionality and power to a Windows form; typically in a project you would use a combination of the two.

This section uses examples that involve adding control to forms because the `Form` control is the container control with which you are most familiar from Chapter 1. You can easily apply these concepts to any of the container controls. A container control has a property named `Controls` that is a collection of `Control` objects. When you add or remove a control from a form, the control is added to or removed from the form's control collection.

EXAM TIP

Adding Control to a Container Control While you are creating a control programmatically, you must remember to associate it with a parent container control. If you don't do this, the control is created but not displayed.

SETTING PROPERTIES OF CONTROLS

Add controls to a Windows form.

- **Set properties on controls.**

Chapter 1 discussed how to work with properties of a `Form` object. You learned how to manipulate these properties through both the Properties window at design time and a program at runtime. The same concept carries over to Windows forms controls. Recall from the section "Using the `System.Windows.Forms.Form` Class" in Chapter 1 that a form is also a control that derives from the `Control` class.

Although this section discusses the properties of controls, some of the properties and their behaviors are similar to those of forms. Controls may also have additional properties, depending on their specific functionality.

To set a property for a control by using the Properties window, follow these steps:

1. Select the control by clicking it. This makes it the active object.
2. Activate the Properties window, and then select a property from the list and modify it.

To set a property for a control within code, follow these steps:

1. Switch to the code view. Select the method or property in which you want to write the code.
2. Use the *`ControlObject`*`.`*`PropertyName`* syntax to access the property for a control's object. *`ControlObject`* is an instance of the control, and *`PropertyName`* is any valid property name for the given control. In the code view, IntelliSense helps you access the list of properties associated with a control.

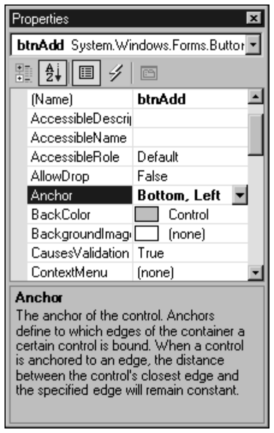
Important Common Properties of Controls

The following sections discuss several important properties that are shared by many of the standard controls.

The `Anchor` Property

When a form is resized, you might want its controls to move along with it, keeping a constant distance from the form's edges.

You can achieve this by anchoring the control with the edges of its container control (see Figures 2.5 and 2.6). The default value of `Anchor` is `Top, Left`; this specifies that the control is anchored to the top and left edge of the container. (Refer to Step by Step 2.4 later in this chapter for an example of how the `Anchor` property is used.)



Properties

btnAdd System.Windows.Forms.Buttor

(Name)	**btnAdd**
AccessibleDescri	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	**Bottom, Left**
BackColor	Control
BackgroundImag	(none)
CausesValidation	True
ContextMenu	(none)

Anchor
The anchor of the control. Anchors define to which edges of the container a certain control is bound. When a control is anchored to an edge, the distance between the control's closest edge and the specified edge will remain constant.

FIGURE 2.5▶
You can click the down arrow in the `Anchor` property to display an anchoring window.

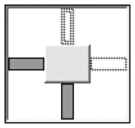
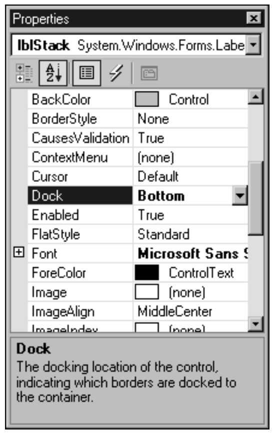


FIGURE 2.6▲
The dark bars indicate the sides to which the control is anchored.

The `Dock` Property

At some point you might need a control to span an entire side (left, right, top, or bottom) of its parent control. You can use the `Dock` property of a control to achieve this behavior (see Figures 2.7 and 2.8). The default value of the `Dock` property is `None`. This property is specially used with controls such as `StatusBar` and `ToolBar`, but it is not limited to them. (To see an example of docking with a `Label` control, see Step by Step 2.4 later in this chapter.)



Properties

lblStack System.Windows.Forms.Labe

BackColor	Control
BorderStyle	None
CausesValidation	True
ContextMenu	(none)
Cursor	Default
Dock	**Bottom**
Enabled	True
FlatStyle	Standard
⊞ Font	**Microsoft Sans S**
ForeColor	ControlText
Image	(none)
ImageAlign	MiddleCenter
ImageIndex	(none)

Dock
The docking location of the control, indicating which borders are docked to the container.

FIGURE 2.7▶
You can click the down arrow in the `Dock` property to display a docking window.

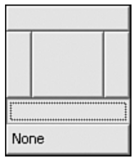


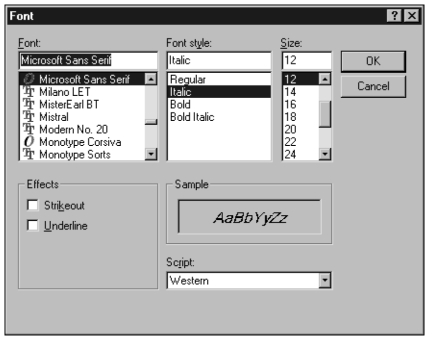
FIGURE 2.8▲
You can click the edge at which you want a control to be docked.

The `Enabled` Property

The `Enabled` property of a control has a Boolean value (`true`/`false`) that can be used to determine whether a control can respond to user interactions. A disabled control (with the `Enabled` property set to `false`) does not receive the focus, does not generate any events, and appears dimmed, or "grayed out." The default value of the `Enabled` property is `true`, except for a `Timer` control, whose `Enabled` property is `false` by default.

The `Font` Property

You use the `Font` property to set the font of the text displayed by the control. The value of this property is an object of the `Font` class. When you select the `Font` property in the Properties window for a control, you see an ellipsis (…) button. Clicking this button invokes a Font dialog box (see Figure 2.9) that can be used to conveniently manipulate the `Font` property.



Font: Microsoft Sans Serif
Font style: Italic
Size: 12
Script: Western

FIGURE 2.9
The Font dialog box allows you to set the `Font` property of a control.

> **NOTE**
>
> **Programmatically Setting the `Font` Property** The `Font` object is said to be *immutable* because its value cannot be modified after the control has been created. If you try to programmatically set one of its properties, you get the compilation error `Property or indexer cannot be assigned to--it is read only`.
>
> Therefore, the only way you can change the `Font` property of a control is by assigning a newly created `Font` object to the `Font` property.

The `Location` Property

The `Location` property specifies the location of the top-left corner of the control with respect to the top-left corner of its container control. Its value is of type `Point`.

Four other properties depend on `Location`: `Left`, `Right`, `Top`, and `Bottom`. `Left` is the same as `Location.X`, `Right` is the same as `Location.X + Width`, `Top` is the same as `Location.Y`, and `Bottom` is the same as `Location.Y + Height`.

The `Name` Property

A control's `Name` property can be used to manipulate a control programmatically. When you place a control on a container object by using the Windows Forms Designer, it names the control automatically based on the type of control (for example, `label1`, `label2`). If you create a control programmatically, its name is by default an empty string. It's a good programming practice to give a meaningful name to a control. Most programmers use Hungarian notation for naming controls; in this scheme, the name of each control begins with a lowercase prefix that is an abbreviation for the name of the control. For example, an instance of a `TextBox` control storing a customer name would be named `txtCustomerName`.

The `Size` Property

The `Size` property sets or gets the height and width of a control. The value of this property is of data type `Size`. `Size` is a struct that has properties named `Height` and `Width` that, respectively, store the height and width of the control, in pixels.

You can also individually manipulate the height and width of a control by using a control's `Height` and `Width` properties.

The `Control` class has a protected property named `DefaultSize` that specifies the default size of the control. The default size is used to draw a control if the size of the control is not explicitly specified. You can override this property in a program to specify a different default size for a control.

The `TabIndex` and `TabStop` Properties

The Tab key is used for keyboard navigation from one control to another on a Windows form. The `TabIndex` property of a control is an integer value that specifies the order in which controls receive focus when the user presses the Tab key.

If you do not want a control to receive focus when the user uses the Tab key, you can set the control's `TabStop` property to `false`. Its default value is `true`, and it allows the control to participate in keyboard navigation through the Tab key.

The `TabIndex` property is effective only when the `TabStop` property of the control is set to `true`.

The `Text` Property

The `Text` property is a string that indicates the text associated with the control. Different controls use the `Text` property in different ways: For example, a `Form` control displays its `Text` property in its title bar and a `Label` control displays its `Text` property on the face of the control. Unlike with a `Form` or a `Label` control, users can manipulate the `Text` property of some controls, such as `TextBox` and `RichTextBox`, at runtime by changing the contents of the controls' input boxes.

The `Text` property can also be used to provide a keyboard shortcut to a control. An ampersand (&) in front of a character marks it as the hotkey for that control. If you assign `&Save` to the `Text` property of a `Button` control, the *S* in the name is underlined. Because this is a standard Windows convention, when users see it, they know that they can press the button by pressing the Alt+S key combination.

Some controls such as `Label` controls cannot receive focus. If you assign a hotkey for such a control, the focus instead goes to a control with the next higher `TabIndex` property. You can in fact use this behavior in your favor. To identify controls such as `TextBox`, `RichTextBox`, `TreeView`, or `ListView`, you would place a `Label` control beside them. You can associate a hotkey with the `Label` control and keep the `TabIndex` property of `Label` and the corresponding control in immediate succession. This way, when you press the hotkey for the `Label` control, it transfers focus to the control that has the next higher `TabIndex` property, and the corresponding control receives focus.

NOTE

Visually Displaying an Ampersand in the `Text` Property What if you want to display an ampersand in a control's text property rather than have it function as a hotkey?

If you're working with a `Label` control, you can set its `UseMnemonic` property to `false`. When `UseMnemonic` is `false`, the control does not interpret the ampersand as a hotkey modifier.

But only the `Label` and `LinkLabel` controls have a `UseMnemonic` property. What about other controls, such as `Button`? With all those other controls, you can use a double ampersand (&&) in the `Text` property to represent a single ampersand.

The `Visible` Property

The `Visible` property is set to `true` by default. When you set it to `false`, you still see the control in the Windows Forms Designer, but users of the application cannot see the control at runtime. Be aware that setting the `Visible` property to `false` does not remove the control from its container's controls collection.

NOTE

Control Transparency No property directly allows you to set transparency for a control. However, you can use the `BackColor` property of a control and set a color by using the `Color.FromArgb()` method. The `Color.FromArgb()` method lets you specify an alpha component that controls transparency.

Configuring the Order of Tabs

Implement navigation for the user interface.

- **Configure the order of tabs**

Many people find it convenient to use the keyboard to navigate among the controls on a form. A Windows user expects to move from one control to another in a logical order by using the Tab key. The Windows Forms Designer provides a Tab Order Wizard that allows you to conveniently set the order in which the controls should receive focus when the Tab key is pressed. Step by Step 2.3 describes how to use this wizard.

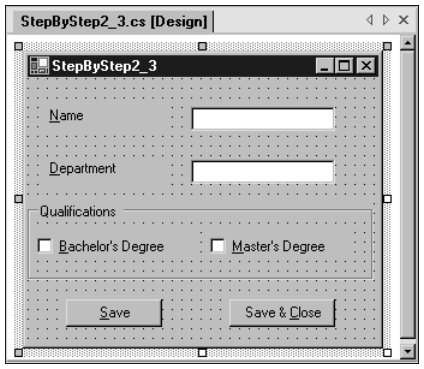


FIGURE 2.10
A hotkey allows you to jump to a control by using the keyboard.

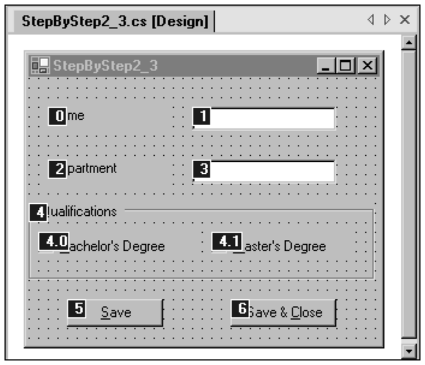


FIGURE 2.11
The Tab Order Wizard allows you to configure the order of tabs.

STEP BY STEP

2.3 Configuring the Order of Tabs

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_3`.
2. Place two `Label` controls on the form and set their `Text` properties to `&Name` and `&Department`.
3. Place two `TextBox` controls and two `CheckBox` controls on the form. Empty the `Text` property for each of the `TextBox` controls. Change the `Text` properties for the `CheckBox` controls as `&Bachelor's degree` and `"&Master's degree`.
4. Add two `Button` controls to the form and change their Text properties to `&Save` and `Save && &Close`. Resize and arrange all the controls as shown in Figure 2.10.
5. Select View, Tab Order and number the controls as shown in Figure 2.11. You can change a tab order number by clicking it.
6. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.
7. Run the project. Use the Tab key to navigate from one control to another. Use hotkeys to directly jump to a control.

HANDLING CONTROL EVENTS

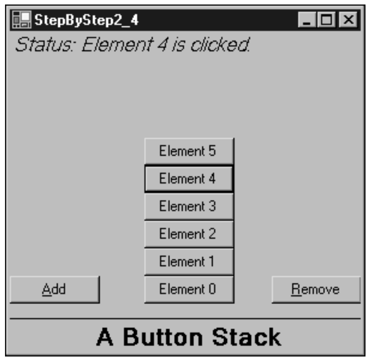
Add controls to a Windows form.

- **Set properties on controls.**
- **Load controls dynamically.**
- **Write code to handle control events and add the code to a control.**

Event handling for a control is very similar to event handling for a Windows form (refer to Chapter 1). Each control inherits a number of events from the `System.Windows.Forms.Control` class.

Each control type also has a set of events that is specific to its unique functionality. Every control has a default event associated with it (for example, the `Click` event for a `Button` control, `Load` for a `Form` control, and `CheckedChanged` for a `CheckBox` control). When you double-click a control in the Windows Forms Designer, the designer automatically creates an event handler for the default event and opens the code view, which allows you to add custom code inside the event handler. You can also handle an event by double-clicking the name of the event in the Properties window; doing this creates an event handler for the selected event of the control.

Step by Step 2.4 is an example of event handling as applied to controls. Step by Step 2.4 creates a Windows form that presents two buttons—Add and Remove. When you click the Add button, the code in its `Click` event handler dynamically adds a new `Button` object to the form, forming a stack of buttons. When you click the Remove button, the code in its `Click` event handler removes the most recently created `Button` object (see Figure 2.12).



Status: Element 4 is clicked.

FIGURE 2.12
You can add and remove controls dynamically.

Step by Step 2.4 illustrates the following points related to handling events:

- How to attach an event handler with a control's event
- How to add custom code to an event handler
- How to attach a single event handler to provide common behavior to several controls
- How to attach an event handler programmatically at runtime

Therefore, you should carefully watch the steps and the comments in the code in Step by Step 2.4.

STEP BY STEP

2.4 Programming Control Events

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_4`.

continues

NOTE

Setting a Property for Multiple Controls To set a property for multiple controls simultaneously, click each of the controls while pressing the Ctrl key; then invoke the Properties window and set the property. The property you set is applied to all the selected controls. When you invoke the Properties window while multiple controls are selected, it shows only the properties that all the selected controls have in common. You can use the same steps to set a common event handler for multiple controls.

continued

2. Place a `Label` control on the form; change the `Dock` property to `Top`, change the `Name` property to `lblStatus`, change the `Font` property to make it italic and size 12, change the `Text` property to be empty, and change the `TextAlign` property to `MiddleCenter`.

3. Place another `Label` control on the form; change the `Dock` property to `Bottom`, change the `Name` property to `lblStack`, change the `Font` property to make it bold and size 16, change the `Text` property to `A Button Stack`, and change the `TextAlign` property to `MiddleCenter`.

4. Place a `Button` control on the form. Change the `Name` property to `btnAdd`, change the `Text` property to `&Add`, and change the `Anchor` property to `Bottom, Left`.

5. Place another `Button` control on the form. Change the `Name` property to `btnRemove`, change the `Text` property to `&Remove`, and change the `Anchor` property to `Bottom, Right`.

6. Switch to the code view and add the following code before the form's constructor code:

```
//Stores the top of stack value
private int intTos;
//stores initial control count
private int intInitCtrlCount;
```

7. Add the following code in the form's constructor code after the call to the `InitializeComponent()` method:

```
//Initially the stack is empty
intTos = 0;
//Get the initial control count. Be sure to put this
//statement after the call to
//InitializeComponent method
intInitCtrlCount = this.Controls.Count;
//Redraw form if it is resized
this.ResizeRedraw=true;
```

8. Using the Properties window, add an event handler for the form's `Paint` event. Add the following code to the event handler:

```
private void StepByStep2_4_Paint(object sender,
    System.Windows.Forms.PaintEventArgs e)
{
    //Gets the Graphics object for the form
    Graphics grfx = e.Graphics;
    //Draw a line, end-to-end on the form
    grfx.DrawLine(Pens.Black, 0,
       this.lblStack.Location.Y -5, this.Width,
       this.lblStack.Location.Y - 5);
    //Set the location for Add and remove buttons
    // so that they get repositioned
    // after the form is resized
    this.btnAdd.Location    = new Point(
      0, this.lblStack.Location.Y - 40);
    this.btnRemove.Location =
        new Point(this.Width-this.btnRemove.Width - 7,
        this.lblStack.Location.Y - 40);
}
```

9. Insert the following event handler in the code; you will later attach it programmatically to the dynamically created `Button` objects:

```
// A custom event handler that I will attach to
// Click event of all buttons added to the stack
private void Button_Click(
    object sender, System.EventArgs e)
{
    //Type cast the object to a Button
    Button btnSender = (Button)sender;
    // Change the lblStatus to show
    // that this button was clicked
    lblStatus.Text = "Status: " +
       btnSender.Text + " is clicked.";
}
```

10. In the design view, double-click the Add button. This attaches an event handler for the `Click` event (the default event for a button). In the event handler code, insert the following lines:

```
private void btnAdd_Click(
    object sender, System.EventArgs e)
{
    //If stack is not yet full
    if (intTos < 8)
    {
        Button btnSender = (Button) sender;
        //Create a new Button to add to the Stack
        Button btnNew = new Button();
        btnNew.Name="Element" + intTos;
        btnNew.Text = "Element " + intTos;
        btnNew.Location =
        new Point((this.Width-btnNew.Width)/2,
    btnSender.Location.Y - btnSender.Height * intTos);
```

continues

continued

```
        //Attach a event handler to the Click
        //event of newly created button
        btnNew.Click += new System.EventHandler(
           this.Button_Click);

        //Add the Button to the Container's
        //Control collection
        this.Controls.Add(btnNew);
        lblStatus.Text = "Status: Element " +
           intTos + " added.";
        intTos++;
    }
    else
        //Stack is full, can't add a button
        lblStatus.Text = "Status: Stack is full!";
}
```

11. In the design view, double-click the Remove button. In its `Click` event handler code, insert the following lines:

```
private void btnRemove_Click(
    object sender, System.EventArgs e)
{
    Button btnSender = (Button) sender;
    //Current control count in the
    //Form's Control collection
    int intCtrlCount = this.Controls.Count;
    //If any new buttons were created in the stack
    if (intCtrlCount > intInitCtrlCount)
    {
        //Remove the most recently added
        //control in the collection
        this.Controls.Remove(
            this.Controls[this.Controls.Count-1]);
        //Adjust the top of stack
        intTos--;
        lblStatus.Text = "Status: Element " +
            intTos + " removed.";
    }
    else
        //Stack is empty, Can't remove a button
        lblStatus.Text = "Status: Stack is empty!";
}
```

12. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.

13. Run the project. Click the Add and Remove buttons. You will see that the `Button` controls are dynamically created and removed. When you click one of the dynamically created button controls, its `Click` event is fired and it displays a message on the top `Label` control (refer to Figure 2.12).

Step by Step 2.4 illustrates these important aspects of dynamic control creation:

- You can dynamically add or remove controls to a container by using the `Add()` and `Remove()` methods of the container object's controls collection.
- You can access the dynamically added control objects by iterating through the controls collection.
- By attaching a single event handler to a group of controls, you can provide the group with a common behavior.

Because event handling is so integrated with the nature of Windows applications, this chapter includes several examples of handling events that are associated with controls.

EXAM TIP

Control Arrays in Visual C# .NET
Visual Basic 6.0 has a concept of control arrays that is often handy when you're working with a group of similar controls. Neither Visual Basic .NET nor Visual C# .NET has any built-in support for control arrays; you can instead achieve the same functionality by manipulating the controls collection of a form or any other container control.

REVIEW BREAK

- You can add controls to a form in two ways: by using the Windows Forms Designer or by hand-coding them in the code.
- The Windows Forms Designer of the Microsoft Visual Studio .NET IDE allows you to add controls to a form and design a form in a very simple manner.
- The Visual Studio .NET toolbox provides a varierty of controls and components to create common Windows GUI elements.
- When you create controls programmatically, be sure to add them to their parent containers' controls collections.
- You can set the properties on controls at design time by using the Properties window or at runtime by accessing them as *`ControlName.PropertyName`* in the code.
- Some of the important properties of the controls `Anchor`, `Dock`, `Enabled`, `Font`, `Location`, `Name`, `Size`, `TabIndex`, `TabStop`, and `Visible` are shared by most common Windows forms controls.
- The Tab Order Wizard provides a convenient way to set the tab order of controls to implement logical keyboard-based navigation of the controls in the form via Tab key.

continues

continued

- Controls are event driven. Events are fired when the user interacts with a control. To cause a control to take specific action when an event occurs, you need to write an event handler method and attach that to an event of a control via the control's delegate.
- You can attach an event handler to a control's event either by using the Properties window or programmatically by adding an event handler's delegate object to a control's event (`ControlName.EventName`) by using the `+=` operator.

GUIDED PRACTICE EXERCISE 2.1

One of the common features of Windows-based applications is that they show or hide controls to make their user interface effective. One common example is the Find and Replace dialog box in Microsoft Word, where controls showing advance options are initially hidden but can be shown if the user wants. How would you create such an interface, where users can control visibility of controls?

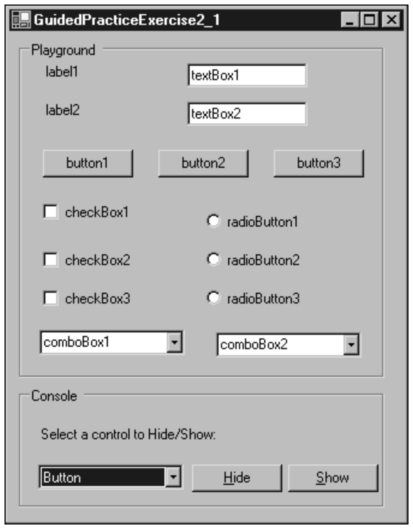


FIGURE 2.13
You can work with the `Controls` collection of a form to hide and show its controls dynamically at runtime.

In this exercise, you will create the Windows form shown in Figure 2.13. The controls on this form are grouped in two `GroupBox` container controls. The Console group box allows you to manipulate the controls in the Playground group box. When you choose a type of control from the combo box in the Console group box and click the Hide button, all controls of that type in the Playground container should be hidden. Similarly, when you click the Show button, the visibility of all controls of the selected type should be restored.

This exercise gives you practice on working with the controls collection of a container control. You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Add a new form to your Visual C# .NET project. Name the form `GuidedPracticeExercise2_1.cs`.

2. Place two `GroupBox` controls on the form. Change the `Name` property of one to `grpConsole` and change the `Name` property of the other to `grpPlayground`. Add and arrange controls on these `GroupBox` controls as shown in Figure 2.13. To easily arrange and align the controls, you can use various options that are available in the Format menu (see Figure 2.14).

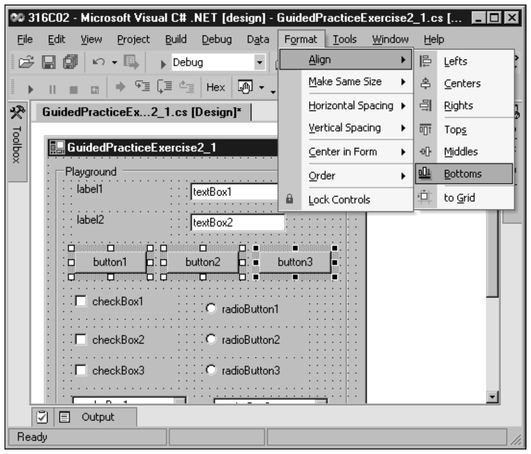
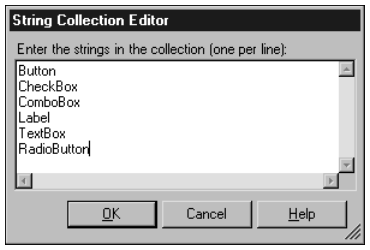


FIGURE 2.14
Format menu options allow you to arrange controls.

3. Name the combo box inside the Console group box `cboControls` and name the buttons `btnHide` and `btnShow`.

4. Invoke the Properties window for the `cboControls` control and select its `Items` property. Click the ellipsis (…) button. This invokes the String Collection Editor (see Figure 2.15). Add the following values to the editor and then close it:

```
Button
CheckBox
ComboBox
Label
TextBox
RadioButton
```


String Collection Editor

Enter the strings in the collection (one per line):

Button
CheckBox
ComboBox
Label
TextBox
RadioButton

FIGURE 2.15
The String Collection Editor allows you to view and change the list of strings for a `ListControl` object such as `ComboBox`.

5. Attach `Click` event handlers with both Hide and Show buttons and enter the following code to manage the `Click` event for them:

```
private void btnShow_Click(
     object sender, System.EventArgs e)
{
```

continues

continued

```
    //check each control in the grpPlayground
    //container control
    foreach (Control ctrl in
        this.grpPlayground.Controls)
    {
        //If the type of control is what selected
        //by user in the combobox
        if (ctrl.GetType().ToString() ==
            "System.Windows.Forms." +
            this.cboControls.SelectedItem)
            //Show the control
            this.grpPlayground.Controls[
this.grpPlayground.Controls.IndexOf(ctrl)].Visible =
            true;
    }
}

private void btnHide_Click(
    object sender, System.EventArgs e)
{
    //check each control in the
    //grpPlayground container control
    foreach (Control ctrl in
        this.grpPlayground.Controls)
    {
        //If the type of control is what
        //selected by user in the combobox
        if (ctrl.GetType().ToString() ==
           "System.Windows.Forms." +
           this.cboControls.SelectedItem)
            //Hide the control
            this.grpPlayground.Controls[
this.grpPlayground.Controls.IndexOf(ctrl)].Visible =
            false;
    }
}
```

6. Insert the `Main()` method and set the form as the startup object for the project.

7. Run the project and experiment with the user interface. You'll find that when you click the Hide button after selecting a control type from the combo box, all controls of that type are hidden from the Playground group box. Similarly, clicking on the Show button displays the control of selected type again.

05 0789728230 CH02 11/21/02 1:20 PM Page 135

If you have difficulty following this exercise, review the sections "Handling Control Events" and "Important Common Properties of Controls" earlier in this chapter. Also, complete Step by Step 2.2 and Step by Step 2.4. Experimenting with those exercises and reading the specified sections should help you relearn this material. After doing that review, try this exercise again.

DIALOG BOXES

A dialog box is used to prompt the user for input. The application can then use the user input for its own processing. You can either use one of the existing dialog box components provided by the Windows Forms library or you can create a dialog box to meet your custom application requirements. The following sections cover both of these scenarios.

Common Dialog Boxes

The Windows Forms library provides the following dialog box classes that are ready to use in Windows applications:

- `ColorDialog`—Displays a list of colors and returns a property that contains the color selected by user.
- `FontDialog`—Displays a dialog box that allows the user to select a font and other text properties, such as size, style, and special effects.
- `OpenFileDialog`—Allows the user to browse files and folders on his or her computer and select one or more files.
- `PageSetupDialog`—Allows the user to select various settings related to page layout.
- `PrintDialog`—Allows the user to select various print-related options and sends specified documents to selected printers.

- `PrintPreviewDialog`—Allows the user to preview a file before printing.
- `SaveFileDialog`—Allows the user to browse the files and folders on his or her computer and select files that need to be saved.

These classes are also referred to as *Windows Forms dialog components*. These dialog boxes provide the same functionality found in several of the common dialog boxes that are used by the Windows operating system. Each of these dialog box classes is derived from the `CommonDialog` class, which provides the basic functionality for displaying a dialog box.

The dialog box classes provide a method named `ShowDialog` that presents a dialog box to the user. Each of the dialog box classes has a set of properties that store data that is relevant to the particular dialog box.

STEP BY STEP

2.5 Using Common Dialog Boxes

1. Add a Windows Form to existing project `316C02`. Name this form `StepByStep2_5`.
2. Place five `Button` controls on the form. Name them `btnOpen`, `btnSave`, `btnClose`, `btnColor`, and `btnFont` and change their `Text` properties to `&Open...`, `&Save...`, `Clos&e...`, `&Color...`, and `&Font...`, respectively.
3. Place a `RichTextBox` control on the form and name it `rtbText`. Arrange all the controls as shown in Figure 2.16.
4. Drag and drop the following components from the toolbox to the form: `OpenFileDialog`, `SaveFileDialog`, `ColorDialog`, and `FontDialog`. Because these are components, they are not added to the form, but they appear on the component tray in the lower area of the form (see Figure 2.16).

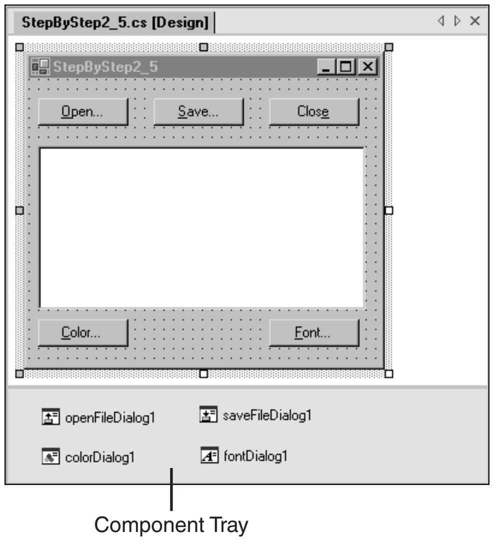


FIGURE 2.16
The component tray represents components that do not otherwise provide visible surfaces at runtime.

NOTE

Nonvisual Controls and the Component Tray Controls such as common dialog box controls do not provide a runtime user interface. Instead of being displayed on the form's surface, they are displayed on a component tray at the bottom of the form. After a control has been added to the component tray, you can select the component and set its properties just as you would with any other control on the form. These nonvisual controls implement the `IComponent` interface and therefore are also sometimes referred to as *components*.

5. Switch to the code view and add the following `using` directive at the top of the program:

```
using System.IO;
```

6. Double-click the Open button to attach an event handler to the `Click` event. Add the following code to the event handler:

```
private void btnOpen_Click(
    object sender, System.EventArgs e)
{
   //Allow to select only *.rtf files
   openFileDialog1.Filter =
      "Rich Text Files (*.rtf)|*.rtf";
   if(openFileDialog1.ShowDialog() == DialogResult.OK)
   {
       //Load the file contents in the RichTextBox
       rtbText.LoadFile(openFileDialog1.FileName,
          RichTextBoxStreamType.RichText);
   }
}
```

7. Add the following code to handle the `Click` event of the Save button:

```
private void btnSave_Click(
    object sender, System.EventArgs e)
{
```

continues

continued

```
   //Default choice to save file is *.rtf
   //but user can select
   //All Files to save with other extension
   saveFileDialog1.Filter =
  "Rich Text Files (*.rtf)|*.rtf|All Files (*.*)|*.*";
   if(saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
       //Save the RichText content to a file
       rtbText.SaveFile(saveFileDialog1.FileName,
          RichTextBoxStreamType.RichText);
    }
}
```

8. Add the following code to handle the `Click` event of the Close button:

```
private void btnClose_Click(
    object sender, System.EventArgs e)
{
    //close the form
    this.Close();
}
```

9. Add the following code to handle the `Click` event of the Color button:

```
private void btnColor_Click(
    object sender, System.EventArgs e)
{
    if(colorDialog1.ShowDialog() == DialogResult.OK)
    {
        //Change the color of selected text
        //If no text selected, change the active color
        rtbText.SelectionColor = colorDialog1.Color;
    }
}
```

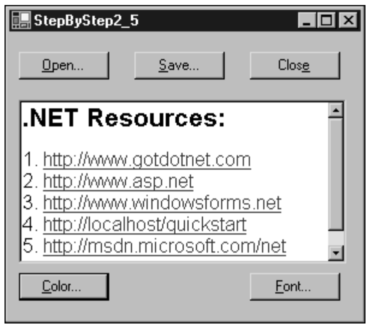
10. Add the following code to handle the `Click` event of the Font button:

```
private void btnFont_Click(
    object sender, System.EventArgs e)
{
    if(fontDialog1.ShowDialog() == DialogResult.OK)
    {
        //Change the font of selected text
        //If no text selected, change the active font
        rtbText.SelectionFont = fontDialog1.Font;
    }
}
```

EXAM TIP

The `FilterIndex` Property The `FilterIndex` property of the `OpenFileDialog` and `SaveFileDialog` components determines the index of the currently selected filter in the list of filters specified by the dialog box's `Filter` property. Be aware that this index is one based; that is, the first filter in the list of filters has an index of one instead of zero.

11. Insert the `Main()` method to launch the form. Set the form as the startup object.
12. Run the project. Click the Open button, select a Rich Text Format (RTF) file to open, experiment with changing the color and font, and save the file (see Figure 2.17).



.NET Resources:

1. http://www.gotdotnet.com
2. http://www.asp.net
3. http://www.windowsforms.net
4. http://localhost/quickstart
5. http://msdn.microsoft.com/net

FIGURE 2.17
The `OpenFileDialog` and `SaveFile` dialog boxes, respectively, allow you to select a file for opening and saving; the `ColorDialog` and `FontDialog` dialog boxes, respectively, allow you to select color and font.

The `Filter` property of the `OpenFileDialog` and `SaveFileDialog` components specifies the choices that appear in the Files of Type drop-down list boxes of these dialog boxes. You can use this property to filter the type of files that the user can select from the dialog box.

You will learn about the printing-related dialog box components in Chapter 11, "Printing."

NOTE

Getting the File Extension The `OpenFileDialog` and `SaveFileDialog` components have a property named `FileName` that returns the name of the selected file. How can you get just the extension for this file? You can do so by using the `Extension` property of the `FileInfo` class:

```
FileInfo fiFileInfo = new
    FileInfo(openFileDialog1.
FileName);
MessageBox.Show(fiFileInfo.
Extension);
```

Similarly if you want just the name of the file, without any extensions, you can use the `Name` property of the `FileInfo` class:

```
FileInfo fiFileInfo = new
    FileInfo(openFileDialog1.
FileName);
MessageBox.Show(fiFileInfo.Name);
```

Creating a Custom Dialog Box

If you need to create dialog boxes other than those already provided by the Windows Forms library, you can do so by creating a form and setting it up to behave as a dialog box. You can make the dialog box as rich as your requirements dictate by adding various controls to it.

STEP BY STEP

2.6 Creating a Custom Dialog Box

1. Add a Windows form to existing project `316C02`. Name the form `frmDialog`.
2. Set the `ControlBox` property of the form to `false`, set `FormBorderStyle` to `FixedDialog`, set `ShowIntaskBar` to `False`, set `StartPosition` to `CenterParent`, and set `Text` to `A Custom Dialog Box`.
3. Place two `Button` controls and a `TextBox` control on the form, set the `Name` property of the `Button` controls to `btnOK` and `btnCancel` and change their `Text` properties to `&OK` and `&Cancel`, respectively. Change the `TextBox` control's `Name` property to `txtDialogText` and its `Text` property to `Dialog Text`.

continues

continued

4. Change the form's constructor code to the following:

```
public frmDialog()
{
    //
    // Required for Windows Forms Designer support
    //
    InitializeComponent();

    //Configure OK button
    btnOK.DialogResult =
       System.Windows.Forms.DialogResult.OK;
    //Configure Cancel button
    btnCancel.DialogResult =
       System.Windows.Forms.DialogResult.Cancel;
}
```

5. Add the following code just after the constructor to create a property that holds the text entered by the user:

```
private string message;
//Stores the message entered by user
public string Message
{
    get
    {
        return message;
    }
    set
    {
        message = value;
    }
}
```

6. Add an event handler for the `Click` event of the OK button and add the following code in it:

```
private void btnOK_Click(
    object sender, System.EventArgs e)
{
    this.Message = this.txtDialogText.Text;
}
```

7. Add a new Windows form to the project. Name it `StepByStep2_6`.

8. Place a `Button` control and a `Label` control on the form. Name the `Button` control `btnInvokeDialog`, and change the `Text` property to `Invoke Dialog`. Name the `Label` control `lblDialogResult`, and change the `Text` property to `Click the button to invoke a custom dialog box`.

NOTE

Modal and Modeless Dialog Boxes When a *modal* dialog box is open, an application can only receive input for the modal dialog box. If you want to work with other windows, you must close the modal dialog box first. An example of a modal dialog box is the one that opens when you select Help, About in Visual Studio .NET. On the other hand, a *modeless* dialog box allows an application to receive input for other windows and controls. An example of a modeless dialog box is the Find & Replace dialog box of Visual Studio .NET.

9. Attach an event handler to the `Click` event of `btnInvokeDialog` and add the following code to it:

```
private void btnInvokeDialog_Click(
    object sender, System.EventArgs e)
{
    //Create the custom dialog box
    frmDialog dlgCustom = new frmDialog();
    //Present dialog box to the user
    dlgCustom.ShowDialog();

    if(dlgCustom.DialogResult == DialogResult.OK)
    {
        //Display the message in label
        //if user pressed OK
        this.lblDialogResult.Text = dlgCustom.Message;
    }
    else
        //Indicate that user cancelled the dialog box
        this.lblDialogResult.Text =
           "Dialog box was cancelled";
}
```

10. Insert a `Main()` method in `StepByStep2_6` to launch the form. Set the form as the startup object for the project.

11. Run the project. Click the Invoke Dialog button, and the custom dialog box is displayed. Enter some text in the text box and click the OK button. The text you enter is then displayed on the parent form's label control (see Figure 2.18).

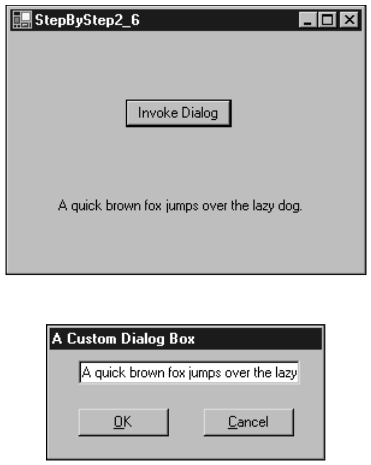

StepByStep2_6
Invoke Dialog
A quick brown fox jumps over the lazy dog.
A Custom Dialog Box
A quick brown fox jumps over the lazy
OK
Cancel

FIGURE 2.18
You can use the `ShowDialog` method to display a form as a dialog box.

The `ShowDialog()` method displays a form as a modal dialog box. All the buttons on the form that need to return results have their `DialogResult` properties set to any of the `DialogResult` enumeration values except `DialogResult.None`. When the user clicks one of these buttons, the button sets the form's `DialogResult` property with the `DialogResult` property of the button and closes the form automatically after running the `Click` event handler (if any).

EXAM TIP

The `Show()` and `ShowDialog()` Methods When you use the `ShowDialog()` method of the `Form` class, the `form` is displayed as a modal dialog box. If you want to display the form as a modeless dialog box, you should use the `Show()` method.

R E V I E W B R E A K

- `DialogBox` is used to prompt the user for input. There are a few built-in dialog boxes available, such as `ColorDialog`, `FontDialog`, `OpenFileDialog`, and `SaveFileDialog`, that function just like the Windows operating system's dialog boxes.

continues

continued

- You can build custom dialog boxes to meet custom requirements. You can create such a dialog box by creating a form and setting a few properties of the form that enable the form to behave like a dialog box.
- Dialog boxes can be of two types: modal and modeless. You call the `ShowDialog()` and `Show()` methods of the `Form` class to create modal and modeless dialog boxes, respectively.

COMMON WINDOWS FORMS CONTROLS

Add controls to a Windows form.

- **Set properties on controls.**
- **Write code to handle control events and add the code to a control.**

The Windows Forms library includes an array of commonly used GUI elements that you can assemble on a Windows form to create Windows applications. These GUI elements (or Windows forms controls) are mostly derived from the `System.Windows.Forms.Control` class. By virtue of this inheritance, these controls share a number of common properties, methods, and events; in addition, these controls may also have their own specific sets of properties, methods, and events that give them distinct behaviors. Figure 2.19 shows a hierarchy of important classes in the `Control` class.

The following sections discuss some important controls that are available in the Windows Forms Designer toolbox. The discussion and examples presented here will help you appreciate the specific nature of these controls.

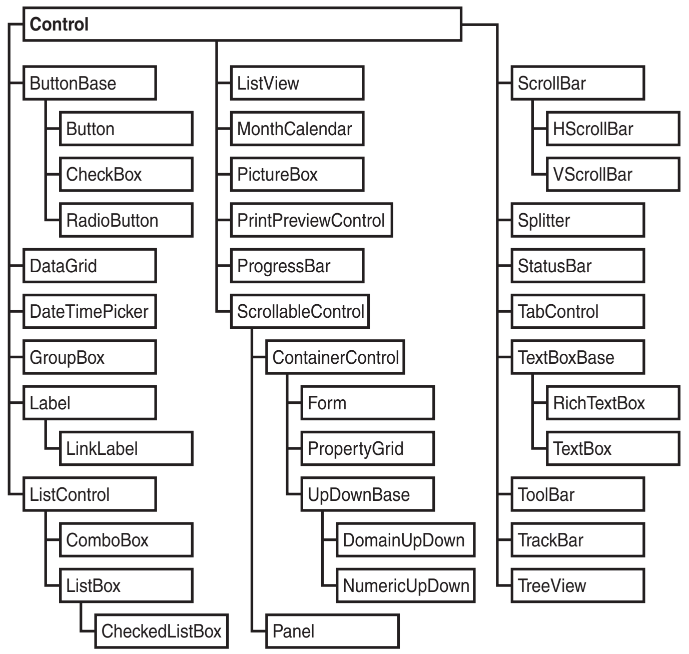


FIGURE 2.19
`System.Windows.Forms.Control` is the base class for all controls.

The `Label` and `LinkLabel` Controls

A `Label` control is used to display read-only information to the user. It is generally used to label other controls and to provide the user with any useful runtime messages or statistics. You can display both text and images on `Label` controls by using the `Text` and `Image` properties, respectively. Table 2.1 shows some of the properties of the `Label` object with which you should be familiar.

TABLE 2.1

IMPORTANT MEMBERS OF THE `Label` CLASS

Member	*Type*	*Description*
`Image`	Property	Specifies an image that is displayed on a label.
`Font`	Property	Specifies the font in which the text is displayed on a label.
`Text`	Property	Specifies the text displayed on a label.
`TextAlign`	Property	Specifies the alignment of the text displayed on a label. It can have one of three horizontal positions (`Center`, `Left`, or `Right`) and one of three vertical positions (`Bottom`, `Middle`, or `Top`).

The `LinkLabel` control is derived from the `Label` control and is very similar to it. However, it has an added functionality: It can also show one or more hyperlinks. Table 2.2 summarizes important properties and events for the `LinkLabel` control.

TABLE 2.2

IMPORTANT MEMBERS OF THE `LinkLabel` CLASS

Member	*Type*	*Description*
`ActiveLinkColor`	Property	Specifies the color used to display an active link.
`DisabledLinkColor`	Property	Specifies the color used to display a disabled link.
`Links`	Property	Gets the collection of `Link` objects in the `LinkLabel` control. The `Link` class contains information about the hyperlink. Its `LinkData` property allows you to associate a uniform resource locator (URL) with the hyperlink.
`LinkArea`	Property	Specifies which portion of text in the `LinkLabel` control is treated as part of the link.
`LinkBehavior`	Property	Specifies how the link appears when the mouse pointer is placed over it.
`LinkClicked`	Event	Occurs when a link in the `LinkLabel` control is clicked. Inside its event handler, the `LinkLabelLinkClickedEventArgs` object provides data for the event. `LinkClicked` is the default event for `LinkLabel` class.
`LinkColor`	Property	Specifies the color used to display a link.
`VisitedLinkColor`	Property	Specifies the color used to display a visited link

STEP BY STEP

2.7 Using `LinkLabel` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_7`.

2. Place two `LinkLabel` controls on the form. Change their `Name` properties to `lnkWinForms` and `lnkPrograms` and their `Text` properties to `Windows Forms Community Website` and `Launch Calculator | Open C: Drive`, respectively.

3. Switch to the code view, and add the following code in the form's constructor, after the `InitializeComponent()` method call:

```
//Add a link for Calculator in
//the first half of LinkLabel
lnkPrograms.Links.Add(
   0, "Launch Calculator".Length, "calc.exe ");
//Add a link for C: Drive in
//the second half of LinkLabel
lnkPrograms.Links.Add(lnkPrograms.Text.IndexOf(
   "Open C: Drive"), "Open C: Drive".Length, "c:\\");

//Autosize the control based on its contents
lnkPrograms.AutoSize = true;
```

4. Double-click the `lnkWinForms` link label to attach a `LinkClicked` event handler to it. Add the following code to the event handler:

```
private void lnkWinForms_LinkClicked(object sender,
System.Windows.Forms.LinkLabelLinkClickedEventArgs e)
{
    lnkWinForms.LinkVisited = true;
    //Go to Windows Forms Community Website
    System.Diagnostics.Process.Start(
       "IExplore", "http://www.windowsforms.net");
}
```

5. Double-click the `lnkPrograms` link label to attach a `LinkClicked` event handler to it. Add the following code to the event handler:

```
private void lnkPrograms_LinkClicked(object sender,
System.Windows.Forms.LinkLabelLinkClickedEventArgs e)
{
    //Launch the program stored in the hyperlink
    System.Diagnostics.Process.Start(
       e.Link.LinkData.ToString());
}
```

6. Insert the `Main()` method to launch form `StepByStep2_7`. Set this form as the startup object for the project.

7. Run the project and click the links. The form takes an appropriate action by either navigating to the Web site, launching the Calculator, or opening the folder (see Figure 2.20).

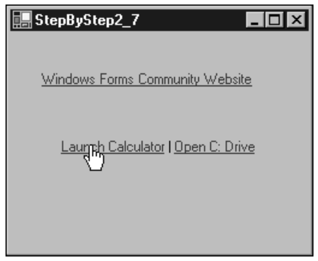

StepByStep2_7
Windows Forms Community Website
Launch Calculator | Open C: Drive

FIGURE 2.20
A `LinkLabel` control can be used to link to a Web page, an application, or a folder.

The `System.Diagnostic.Process` class lets you start and stop processes that are running on your computer. Its `Start()` method actually starts a given process. The `Start()` method is static and therefore lets you create the process without creating an object of `System.Diagnostic.Process` class. The `LinkLabelLinkClickedEventArgs` object passed to the `LinkClicked` event handler contains a `Link` object that corresponds to the link being clicked. The `LinkData` property of this `Link` object represents the data associated with the link.

The `TextBox` and `RichTextBox` Controls

`TextBox` and `RichTextBox` both derive from the `TextBoxBase` class. The `TextBoxBase` class implements the basic functionality used by both the `TextBox` and `RichTextBox` classes.

A `TextBox` control provides an area that the user can use to input text. Depending on how you set the properties of this control, you can use it for multiline text input or you can use it like a password box that masks the characters entered by the user with a specific character (such as `*`). Table 2.3 summarizes the important members of the `TextBox` class.

TABLE 2.3

IMPORTANT MEMBERS OF THE `TextBox` CLASS

Member	*Type*	*Description*
`AcceptReturn`	Property	Represents a Boolean value, where `true` indicates that pressing the Enter key in a multiline text box inserts a new line. This property is applicable only if the text box accepts multiline input.
`CharacterCasing`	Property	Specifies whether the `TextBox` control needs to modify the case of the characters as they are entered. The value of this property can be `Lower`, `Normal`, or `Upper`. The Default value is `Normal`, which means the characters are not modified.
`MultiLine`	Property	Indicates whether the `text box` can accept multiple lines of input. The default value is `false`.

Member	*Type*	*Description*
`PasswordChar`	Property	Masks each character in the text box by the specified character. It is usually set when the text box inputs sensitive information such as a password, where the characters need to be masked. If no character is specified, the normal text is displayed.
`ReadOnly`	Property	Makes the text box appear with a gray background and text cannot be edited when set to `true`.
`ScrollBars`	Property	Specifies which scrollbars (none, horizontal, vertical, or both) should appear in a multiline textbox.
`Text`	Property	Specifies the text contained in the textbox.
`TextChanged`	Event	Occurs when the value of the `Text` property changes. `TextChanged` is the default event for the `TextBox` class.
`WordWrap`	Property	Specifies whether the control can automatically wrap words to the next line. The default value is `true`. Works only if the `MultiLine` property is set to `true`.

As its name suggests, the `RichTextBox` control is a `TextBox` control with rich formatting capabilities. It can upload an RTF file. It can display its contents in rich character and paragraph formatting. Any portion of the control can be displayed in various formats, depending on the settings of its properties. Table 2.4 summarizes the important members of the `RichTextBox` class.

TABLE 2.4

IMPORTANT MEMBERS OF THE `RichTextBox` CLASS

Member	*Type*	*Description*
`DetectUrls`	Property	Specifies whether the control automatically detects and formats URLs
`Rtf`	Property	Specifies the text of the `RichTextBox` control, including all RTF codes
`SelectionColor`	Property	Specifies the color of the currently selected text
`SelectionFont`	Property	Specifies the font of the currently selected text

continues

TABLE 2.4 *continued*

IMPORTANT MEMBERS OF THE RichTextBox CLASS

Member	*Type*	*Description*
`SelectedRtf`	Property	Specifies the currently selected RTF text
`TextChanged`	Event	Occurs when the value of the `Text` property changes. `TextChanged` is the default event for `RichTextBox` class.
`WordWrap`	Property	Specifies whether the control can automatically wrap words to the next line if required
`ZoomFactor`	Property	Specifies the current zoom level.

Step by Steps 2.5 and 2.18, later in this chapter, describe good usage of the `RichTextBox` control.

The `PictureBox` Control

`PictureBox` controls display images and graphics from metafile, icon, bitmap, JPEG, PNG, and GIF files. Table 2.5 summarizes the important members of the `PictureBox` class.

TABLE 2.5

IMPORTANT MEMBERS OF THE PictureBox CLASS

Member	*Type*	*Description*
`Click`	Event	Occurs when the control is clicked. `Click` is the default event for the `PictureBox` class.
`Image`	Property	Represents the image that the picture box displays.
`SizeMode`	Property	Indicates how the image is displayed. Holds one of the `PictureBoxSizeMode` enumeration values—`AutoSize` (picture box is autosized to the image size), `CenterImage` (image is displayed in the center of the picture box), `Normal` (image is placed in the upper-left corner of the picture box), and `StretchImage` (image is stretched or reduced to fit the picture box size).

Step by Steps 2.8 and 2.16, later in this chapter, show good usage of `PictureBox` control.

The `GroupBox` and `Panel` Controls

`GroupBox` is a container control that contains other controls. It is mostly used to arrange controls and group similar controls. It does not include scrollbars. Table 2.6 summarizes the important members of the `GroupBox` class.

TABLE 2.6

IMPORTANT MEMBERS OF THE `GroupBox` CLASS

Member	*Type*	*Description*
`Controls`	Property	Specifies a collection of controls contained in a group box
`Text`	Property	Specifies a caption for a group box

Similar to `GroupBox`, `Panel` is a container control that contains other controls. It is mostly used to arrange controls and group similar controls. It has built-in support for scrollbars. You cannot provide a caption for the `Panel` control. Table 2.7 summarizes the important members of the `Panel` class.

TABLE 2.7

IMPORTANT MEMBERS OF THE `Panel` CLASS

Member	*Type*	*Description*
`AutoScroll`	Property	Indicates whether scrollbars should be displayed when the display of all the controls exceeds the area of a panel
`Controls`	Property	Specifies a collection of controls contained in a panel

STEP BY STEP

2.8 Using GroupBox and Panel Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_8`.

2. Place on the form a `Label` control with the `Text` property set to `Click button to open a picture file:` and a `Button` control with the `Text` property set to `Browse...` and the `Name` property set to `btnBrowse`. Also, add to the form an `OpenFileDialog` control with the `Name` property set to `ofdPicture`.

3. Place a `GroupBox` control on the form and add three label controls to it. Name the three label controls `lblSize`, `lblDateModified`, and `lblDateAccessed`. Set the `GroupBox` control's `Name` property to `grpFile` and `Text` property to `File Statistics`.

4. Place a `Panel` control in the form and add a `PictureBox` control to it. Set the `Panel` control's `Name` property to `pnlImage` and `AutoScroll` property to `true`. Set the `PictureBox` control's `Name` property to `pbImage` and `SizeMode` property to `AutoSize`.

5. Switch to the code view and add the following `using` directive at the top of the program:

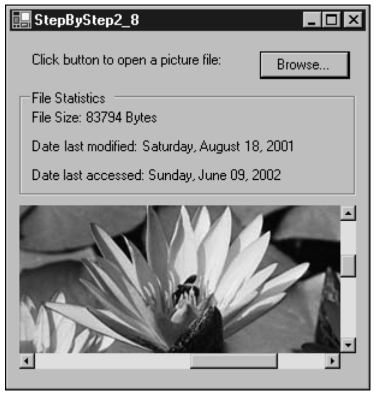
```
using System.IO;
```

6. Double-click the `btnBrowse` button to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void btnBrowse_Click(
    object sender, System.EventArgs e)
{
    //Set filters for graphics files
    ofdPicture.Filter=
      "Image Files (BMP, GIF, JPEG, etc.)|" +
      "*.bmp;*.gif;*.jpg;*.jpeg;*.png;*.tif;*.tiff|" +
      "BMP Files (*.bmp)|*.bmp|" +
      "GIF Files (*.gif)|*.gif|" +
      "JPEG Files (*.jpg;*.jpeg)|*.jpg;*.jpeg|" +
      "PNG Files (*.png)|*.png|" +
      "TIF Files (*.tif;*.tiff)|*.tif;*.tiff|" +
      "All Files (*.*)|*.*";
    if(ofdPicture.ShowDialog() == DialogResult.OK)
    {
        //Get file information
        FileInfo file = new FileInfo(
           ofdPicture.FileName);
        lblSize.Text = String.Format(
            "File Size: {0} Bytes",
            file.Length.ToString());
        lblDateModified.Text = String.Format(
           "Date last modified: {0}",
            file.LastWriteTime.ToLongDateString());
        lblDateAccessed.Text = String.Format(
            "Date last accessed: {0}",
            file.LastAccessTime.ToLongDateString());
        //Load the file contents in the PictureBox
        this.pbImage.Image = new Bitmap(
           ofdPicture.FileName);
    }
}
```

7. Insert the `Main()` method to launch form `StepByStep2_8`. Set this form as the startup object for the project.

8. Run the project. Click the Browse button. The Open dialog box prompts you to open an image file. Select an appropriate image file and click OK. The `Panel` control shows the image, and the `GroupBox` control shows the file statistics (see Figure 2.21). The `Panel` control includes scrollbars if the image size exceeds the panel area.



File Size: 83794 Bytes

Date last modified: Saturday, August 18, 2001

Date last accessed: Sunday, June 09, 2002

FIGURE 2.21
The `Panel` and `GroupBox` controls show the image and file statistics.

The `Button`, `CheckBox`, and `RadioButton` Controls

A `Button` object is used to initiate a specific action when a user clicks it. The `Button` class derives from the `ButtonBase` class. The `ButtonBase` class provides common functionality to the `Button`, `CheckBox`, and `RadioButton` classes.

In contrast to the `Button` class, `CheckBox` and `RadioButton` are used to maintain state. They can be on or off (that is, selected or not selected, checked or unchecked). These controls are generally used in groups. A `CheckBox` control allows you to select one or more options from a group of options, and a group of `RadioButton` controls are used to select one out of several mutually exclusive options. If you want to place two groups of `RadioButton` controls on a form and have each group allow one selection, you need to place them in different container controls, such as `GroupBox` or `Panel` controls, on the form.

These container controls, as discussed earlier in the chapter, are used to group controls that have similar functionality. The `GroupBox` control is a popular choice for grouping `RadioButton` controls.

Tables 2.8, 2.9, and 2.10 summarize the important members of the `Button`, `CheckBox`, and `RadioButton` classes, respectively.

TABLE 2.8

IMPORTANT MEMBERS OF THE `Button` CLASS

Member	*Type*	*Description*
`Image`	Property	Specifies the image displayed on a button.
`Text`	Property	Specifies the text displayed on a button.
`Click`	Event	Occurs when the `Button` control is clicked. `Click` is the default event for the `Button` class.

TABLE 2.9

IMPORTANT MEMBERS OF THE `CheckBox` CLASS

Member	*Member*	*Description*
`Checked`	Property	Returns `true` if the check box has been checked. Otherwise, it returns `false`.
`CheckedChanged`	Event	Occurs every time a check box is checked or unchecked. `CheckedChanged` is the default event for the `CheckBox` class
`CheckState`	Property	Specifies the state of the check box. Its value is one of the three `CheckState` enumeration values: `Checked`, `Unchecked`, or `Indeterminate`.
`ThreeState`	Property	Indicates whether the check box allows three states: `Checked`, `Unchecked`, or `Indeterminate`. If it is set to `false`, `CheckState` can be set to `Indeterminate` only in code and not through the user interface.
`Text`	Property	Specifies the text displayed along with the check box.

EXAM TIP

A Checked Property Doesn't Always Indicate the Checked State If the `ThreeState` property of a `CheckBox` control is true, the `Checked` property returns `true` for `Checked` as well as for the `Indeterminate` check state. Therefore, the `CheckState` property should be used to determine the current state of the check state.

TABLE 2.10

IMPORTANT MEMBERS OF THE RadioButton CLASS

Member	*Type*	*Description*
`Checked`	Property	Indicates whether the radio button is selected. Returns `true` if the button is selected and `false` otherwise.
`CheckedChanged`	Event	Occurs every time the control is either selected or deselected. `CheckedChanged` is the default event of the `RadioButton` class.
`Text`	Property	Specifies the text displayed along with the radio button.

STEP BY STEP

2.9 Using `CheckBox` and `RadioButton` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_9`.
2. Add three `GroupBox` controls to the form. Change their `Name` properties to `grpSampleText`, `grpEffects`, and `grpFontSize`. To `grpSampleText`, add a `Label` control, and then add two `CheckBox` controls to `grpEffects` and three `RadioButton` controls to `grpFontSize`. Arrange the controls and change their `Text` properties as shown in Figure 2.22.
3. Change the `Name` property of the `Label` control to `lblSampleText`. Change the `Name` properties of the two `CheckBox` controls to `cbStrikeout` and `cbUnderline`. Change the `Name` properties of `RadioButton` controls to `rb12Point`, `rb14Points`, and `rb16Points`.
4. Double-click the `CheckBox` and `RadioButton` controls and add the following code to the default event handlers:

```
private void cbStrikeout_CheckedChanged(
    object sender, System.EventArgs e)
{
    //toggle the Strikeout FontStyle of lblSampleText
    lblSampleText.Font = new Font(
     lblSampleText.Font.Name, lblSampleText.Font.Size,
     lblSampleText.Font.Style ^ FontStyle.Strikeout);
}
```

continues

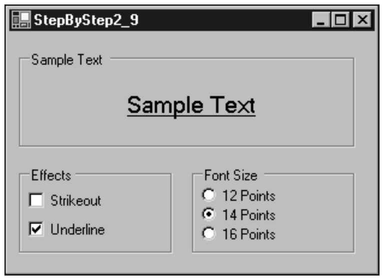


FIGURE 2.22
You use the `CheckBox` control to select a combination of options and the `RadioButton` control to select one of several mutually exclusive options.

NOTE

The `AutoCheck` Property When the `AutoCheck` property of a `CheckBox` or a `RadioButton` control is `true`, the `Checked` property (even the `CheckState` property, in the case of a `CheckBox` control) and the appearance of the control are automatically changed when the user clicks the control. You can set the `AutoCheck` property to `false` and then write code in the `Click` event handler to have these controls behave in a different manner.

EXAM TIP

The `FontStyle` Enumeration and Bitwise Operations The `FontStyle` enumeration has a `Flags` attribute that allows bitwise operations on `FontStyle` values. For example, look at the following statement:

```
lblSampleText.Font.Style |
FontStyle.Underline
```

Here the `|` operator will turn on all the bits representing the Underline style, returning a `FontStyle` value that adds Underline to the existing font style of `lblSampleText`.

The following expression involves a bitwise exclusive `OR` (`XOR`) operation:

```
lblSampleText.Font.Style ^
FontStyle.Underline
```

This expression returns a `FontStyle` value that toggles the `Underline` font style of the label. If the label was already underlined, the new value has the underline removed; if the label was not underlined already, the `Underline` bits are set in the new value.

The following expression involving a bitwise `AND` does not have any effect because using `AND` with 1 always returns the original value:

```
lblSampleText.Font.Style &
FontStyle.Underline
```

continued

```
private void cbUnderline_CheckedChanged(
    object sender, System.EventArgs e)
{
    //toggle the Underline FontStyle of lblSampleText
    lblSampleText.Font = new Font(
     lblSampleText.Font.Name, lblSampleText.Font.Size,
     lblSampleText.Font.Style ^ FontStyle.Underline);
}

private void rb12Points_CheckedChanged(
   object sender, System.EventArgs e)
{
    //Change the font size of lblSampleText to 12
    lblSampleText.Font = new Font(
     lblSampleText.Font.Name, 12,
     lblSampleText.Font.Style);
}

private void rb14Points_CheckedChanged(
   object sender, System.EventArgs e)
{
    //Change the font size of lblSampleText to 14
    lblSampleText.Font = new Font(
       lblSampleText.Font.Name, 14,
       lblSampleText.Font.Style);
}

private void rb16Points_CheckedChanged(
    object sender, System.EventArgs e)
{
    //Change the font size of lblSampleText to 16
    lblSampleText.Font = new Font(
       lblSampleText.Font.Name, 16,
       lblSampleText.Font.Style);
}
```

5. Insert a `Main()` method to launch the form `StepByStep2_9`. Set the form as the startup object for the project.

6. Run the project. Click the `CheckBox` controls, and you see that the font style of sample text changes. Then click the `RadioButton` controls, and you are able to select only one of three radio buttons, and when you click one, the `CheckedChanged` event handler immediately increases or decreases the font size (refer to Figure 2.22).

The `ListBox`, `CheckedListBox`, and `ComboBox` Controls

A `ListBox` control allows you to select one or more values from a given list of values. It derives from the `ListControl` class, which provides common functionality for both the `ListBox` and `ComboBox` controls. Table 2.11 summarizes the important members of the `ListBox` class.

TABLE 2.11

IMPORTANT MEMBERS OF THE `ListBox` CLASS

Member	*Type*	*Description*
`ColumnWidth`	Property	Specifies the width of a column in a multi-column list box.
`ItemHeight`	Property	Specifies the height of an item in a list box.
`Items`	Property	Specifies a collection of objects representing the list of items in a list box.
`FindString()`	Method	Finds the first item in a list box that starts with the specified string.
`FindStringExact()`	Method	Finds the first item in a list box that exactly matches the specified string.
`MultiColumn`	Property	Indicates whether a list box supports multiple columns.
`SelectedIndex`	Property	Specifies an index of the currently selected item.
`SelectedIndexChanged`	Event	Occurs when the selected index property changes. `SelectedIndexChanged` is the default event for the `ListBox` class.
`SelectedIndices`	Property	Specifies a collection of indexes of the currently selected items.
`SelectedItem`	Property	Specifies a currently selected item.
`SelectedItems`	Property	Specifies a collection of currently selected items.

continues

TABLE 2.11 *continued*

IMPORTANT MEMBERS OF THE ListBox CLASS

Member	*Type*	*Description*
`SelectionMode`	Property	Indicates the number of items that can be selected. The values are specified by the `SelectionMode` enumeration and can be `MultiSimple` (allows multiple selections), `MultiExtended` (allows multiple selections, with the help of the Ctrl, Shift, and arrow keys), `None` (allows no selection), and `One` (allows a single selection).
`Sorted`	Property	Indicates whether the items in a list box are sorted alphabetically.

The `CheckedListBox` control derives from the `ListBox` control and inherits most of the features of the `ListBox` class. A `CheckedListBox` control displays a list of items to be selected, along with a check box for each item. The user selects an item by clicking the check box associated with the item. Because a `CheckedListBox` control contains check boxes, it implies that zero or more items can be selected from a `CheckedListBox` control. Table 2.12 summarizes important members of the `CheckedListBox` class.

NOTE

The `SelectionMode` Property of the `CheckedListBox` Class Unlike the `ListBox` class, the `CheckedListBox` class allows only two values of the `SelectionMode` enumeration for its `SelectionMode` property. The value `None` does not allow any selection, and the value `One` allows you to make zero or more selections. It is invalid to use the other values of the `SelectionMode` enumeration, such as `MultiSimple` and `MultiExtended`, with the `SelectionMode` property of the `CheckedListBox` control.

TABLE 2.12

IMPORTANT MEMBERS OF THE CheckedListBox CLASS

Member	*Type*	*Description*
`CheckedIndices`	Property	Specifies a collection of indexes of the currently checked items.
`CheckedItems`	Property	Specifies a collection of currently checked items.
`ItemCheck`	Event	Occurs when an item is checked or unchecked.
`SelectionMode`	Property	Indicates the number of items that can be checked. The values are specified by the `SelectionMode` enumeration and can be only `None` (allow no selection) or `One` (allow multiple selections).

STEP BY STEP

2.10 Using `ListBox` and `CheckedListBox` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_10`.
2. Add two `Label` controls, a `CheckedListBox` control, a `ListBox` control, and a `Button` control, and arrange them as shown in Figure 2.23. Change the `Label` control's Text properties to `Select Scripts` and `Selected Scripts`. Change the `Button` control's `Name` property to `btnDone` and change its `Text` property to `Done`.
3. Change the `ListBox` control's `Text` property to `lbSelectedScripts` and `SelectionMode` to `MultiExtended`. Name the `CheckedListBox` control `clbScripts`, select its `Items` property, and click the ellipsis (…) button. Add the following scripts in the String Collection Editor:

```
Latin
Greek
Cyrillic
Armenian
Hebrew
Arabic
Devanagari
Bengali
Gurmukhi
Gujarati
Oriya
Tamil
Telugu
Kannada
Malayalam
Thai
Lao
Georgian
Tibetan
Japanese Kana
```

4. Switch to the code view and add the following `using` directive:

```
using System.Text;
```

continues

continued

5. Invoke the Properties window and click the Events icon. Double-click the `ItemCheck` event to add an event handler for the event. Add the following code to the event handler:

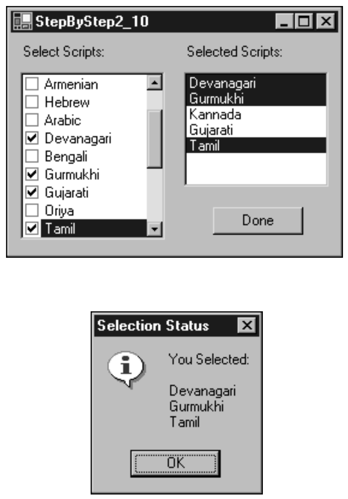
```
private void clbScripts_ItemCheck(object sender,
    System.Windows.Forms.ItemCheckEventArgs e)
{
    //Get the item that was just checked or unchecked
    string item = clbScripts.SelectedItem.ToString();
    if (e.NewValue == CheckState.Checked)
        //Checked: Add to the ListBox
        lbSelectedScripts.Items.Add(item);
    else
        //Unchecked: Remove from the ListBox
        lbSelectedScripts.Items.Remove(item);
}
```

6. Double-click the `btnDone` control and add the following code to handle the `Click` event of the `Button` control:

```
private void btnDone_Click(
    object sender, System.EventArgs e)
{
   //Be sure to have a using directive for System.Text
   //at top of the program
   StringBuilder sbLanguages = new StringBuilder();
   if (lbSelectedScripts.SelectedItems.Count>0)
   {
       sbLanguages.Append("You Selected:\n\n");
       //If there were items selected in ListBox
       //create a string of their names
       foreach (string item in
           lbSelectedScripts.SelectedItems)
           sbLanguages.Append(item + "\n");
   }
   else
   {
       //No items selected
       sbLanguages.Append(
          "No items selected from ListBox");
   }
   MessageBox.Show(sbLanguages.ToString(),
      "Selection Status",
      MessageBoxButtons.OK,
      MessageBoxIcon.Information);
}
```

7. Insert the `Main()` method to launch the form `StepByStep2_10`. Set the form as the startup object for the project.

8. Run the project. Double-click the `CheckBox` control to select items from the `CheckedListBox` control. The selected scripts are then added to the `ListBox` control. Select some items from the `ListBox` control and then click the Done button. A message box is displayed, showing the selected scripts from the `ListBox` control (see Figure 2.23).



You Selected:

Devanagari
Gurmukhi
Tamil

FIGURE 2.23
The `ListBox` and `CheckedListBox` controls allow the user to select a combination of values from a list of items.

A `ComboBox` control is similar to a `ListBox` control, except that it has an editing field. A combo box appears with an editing text box with a down arrow at the right side of the box. When the down arrow is clicked, a drop-down list containing the predefined items to be displayed by the combo box appears. You can select only a single item from the combo box. A `ComboBox` control allows you to enter new text or select from the list of existing items in the combo box. Table 2.13 summarizes the important members of the `ComboBox` class with which you should be familiar.

TABLE 2.13

IMPORTANT MEMBERS OF THE `ComboBox` CLASS

Member	*Type*	*Description*
`DrawMode`	Property	Specifies how combo box items are drawn. It has one of the values from the `DrawMode` enumeration, `Normal`, which specifies that the list of items is drawn by the system itself. The other two values, `OwnerDrawFixed` and `OwnerDrawVariable`, specify that the elements are drawn by your own program (preferably in the `DrawItem` event handler). `OwnerDrawFixed` specifies that elements be of the same size, and `OwnerDrawVariable` specifies a variable size.

continues

TABLE 2.13 *continued*

IMPORTANT MEMBERS OF THE ComboBox CLASS

Member	*Type*	*Description*
`DropDownStyle`	Property	Represents the style of the combo box. Its values are specified by the `DropDownStyle` enumeration values `DropDown` (default style, click the arrow button to display the items, and the text portion is editable), `DropDownList` (click the arrow button to display the items, but the text portion is not editable), and `Simple` (no arrow button, the list portion is always visible, and the text portion is also editable).
`DropDownWidth`	Property	Specifies the width of the drop-down list portion of the combo box.
`Items`	Property	Specifies a collection of items in the combo box control.
`MaxDropDownItems`	Property	Represents the maximum number of items the drop-down list portion can display. If the number of items is greater than this property, a scrollbar appears.
`MaxLength`	Property	Indicates the maximum length of text allowed to be entered in the editable portion of the combo box.
`SelectedIndex`	Property	Specifies an index of the currently selected item.
`SelectedIndexChanged`	Event	Occurs when the `SelectedIndex` property changes. `SelectedIndexChanged` is the default event for the `ComboBox` class.
`SelectedItem`	Property	Specifies the currently selected item.
`SelectedText`	Property	Specifies the currently selected text in the editable portion.
`Sorted`	Property	Indicates whether the items are sorted alphabetically in the combo box.

EXAM TIP

The `SelectedIndex` Property The `SelectedIndex` property in the `ListBox`, `CheckedListBox`, and `ComboBox` controls returns `-1` if no item is selected.

STEP BY STEP

2.11 Using ComboBox Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_11`.

2. Place a `Label` control with the `Text` property `Select or Enter a Color`, a `ComboBox` control with the `Name` property `cboColor`, and a `Button` control with the `Name` property `btnSet` and the `Text` property `Set Form's Back Color`.

3. Change the `ComboBox` control's `Sorted` property to `true` and add the following scripts to the items collection via the String Collection Editor:

```
Violet
Indigo
Blue
Green
Yellow
Orange
Red
White
```

4. Double-click the `btnSet` control and add the following code to handle the `Click` event of the `Button` control:

```
private void btnSet_Click(
    object sender, System.EventArgs e)
{
    this.BackColor = Color.FromName(cboColor.Text);
}
```

5. Insert a `Main()` method to launch the form `StepByStep2_11`. Set the form as the startup object for the project.

6. Run the project. Select a color from the list of colors in the combo box or enter a new color in the combo box. Click the button. The form's background color is changed to the color that is selected or entered in the combo box. Figure 2.24 shows the output when a desired color is entered in the combo box and the button is clicked.

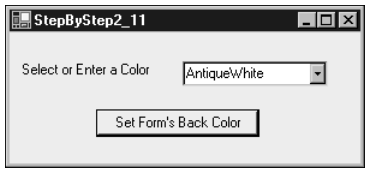


FIGURE 2.24
The `ComboBox` control allows you to either select from a list or enter new text.

GUIDED PRACTICE EXERCISE 2.2

Some Windows-based applications use sophisticated combo boxes that have rich user interfaces that can display things such as images and text in various fonts. Ever wonder how you would do such customization in applications?

In this exercise you will create the Windows form shown in Figure 2.25. The idea is to create a font sampler such as those used by many word processing applications. The form contains a combo box that displays a list of fonts installed on the system. But the interesting thing here is that the items in the combo box are displayed in their respective fonts. The form also contains a label control that displays a sample of the font that the user chooses from the combo box.

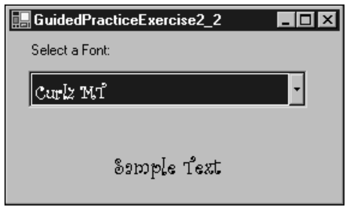


FIGURE 2.25
You can create an owner-drawn `ComboBox` control by programming the `DrawItem` event handler.

This exercise gives you practice on working with controls that have customized rendering. If you are looking for a starting point, try experimenting with the `DrawMode` property and the `DrawItem` event of the `ComboBox` control and then proceed with this exercise. You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Add a new form to your Visual C# .NET project. Name the form `GuidedPracticeExercise2_2.cs`.
2. Place two `Label` controls on the form—one with the `Text` property `Select a Font` and the other with the `Name` property `lblSampleText`, the `Text` property `Sample Text`, and the `Font` property `Microsoft Sans Serif, Regular, 14`.
3. Place a `ComboBox` control on the form. Change the `Name` property to `cboFont`, change `DrawMode` to `OwnerDrawVariable`, and change `DropDownStyle` to `DropDownList`.
4. Double-click the form and add the following code to handle the `Load` event of the form:

```
private void GuidedPracticeExercise2_2_Load(
    object sender, System.EventArgs e)
{
    //Add a list of System Fonts to ComboBox
    cboFont.Items.AddRange(FontFamily.Families);
}
```

5. Add the following code in the code view:

```
private FontStyle GetFontStyle(FontFamily ff)
{
    FontStyle fontStyle = FontStyle.Regular;
    // Check whether Regular style is available
    if (!ff.IsStyleAvailable(FontStyle.Regular))
        fontStyle = FontStyle.Italic;
    // Check whether Italic style is available
    if (!ff.IsStyleAvailable(FontStyle.Italic))
        fontStyle = FontStyle.Bold;
    return fontStyle;
}
```

6. Invoke the Properties window, click the Events icon, select the `DrawItem` event, and double-click to add an event handler to `DrawItem`. Add the following code to the event handler:

```
// This DrawItem event handler is invoked
// to draw an item in a ComboBox if that
// ComboBox is in an OwnerDraw DrawMode.
private void cboFont_DrawItem(object sender,
    System.Windows.Forms.DrawItemEventArgs e)
{
    ComboBox cboFont = (ComboBox) sender;
    // do nothing if there is no data
    if (e.Index == -1)
        return;
    if (sender == null)
        return;

    //Make a FontFamily object from the name
    //of the font currently being drawn
    FontFamily fontFamily =
       (FontFamily) cboFont.Items[e.Index];

    // Create a Font object that will be used
    // to draw the text in ComboBox
    Font font  = new Font(fontFamily.Name, 12,
      GetFontStyle(fontFamily));

    // If the item is selected,
    // draw the correct background color
    e.DrawBackground();
    e.DrawFocusRectangle();

    // DrawItemEventArgs gives access to
    // the ComboBox Graphics object
    Graphics g = e.Graphics;

    // Draw the name of the font in the same font
    g.DrawString(fontFamily.Name, font,
      new SolidBrush(e.ForeColor),
      e.Bounds.X, e.Bounds.Y+4);
}
```

continues

continued

7. Double-click the combo box and add the following code to handle the `SelectedIndexChanged` event of the `ComboBox` control:

```
private void cboFont_SelectedIndexChanged(
    object sender, System.EventArgs e)
{
    // Get the FontFamily object for
    // current ComboBox selection
    FontFamily fontFamily =
       (FontFamily)((ComboBox) sender).SelectedItem;
    // Create a Font object and draw the Font Name
    lblSampleText.Font = new Font(fontFamily.Name,
   lblSampleText.Font.Size, GetFontStyle(fontFamily));
}
```

8. Insert the `Main()` method to launch the form and set the form as the startup object for the project.

9. Run the project. The combo box displays all the available fonts, drawn in their own fonts. The label's text is also updated to display the text in the selected font (refer to Figure 2.25).

If you have difficulty following this exercise, review the section "The `ListBox`, `CheckedListBox` and `ComboBox` Controls," earlier in this chapter. After doing that review, try this exercise again

REVIEW BREAK

- The `LinkLabel` control is derived from the `Label` control. It allows you to add links to a control. The `Links` property of the `LinkLabel` control contains a collection of all the links referenced by the control.

- You can display a `TextBox` control as an ordinary text box, a password text box (in which each character is masked by the character provided in the `PasswordChar` property), or a multiline text box by setting the `TextBox` control's `MultiLine` property to `true`. The `RichTextBox` control provides enriched formatting capabilities to a text box control. It can also be drawn as a single-line or multiline text box. By default, it has its `MultiLine` property set to `true`, unlike the `TextBox` control.

- `GroupBox` and `Panel` controls are container controls. They can be used to group similar controls. The `Controls` property of these controls contains a collection of the controls' child controls.
- The `CheckBox` control allows multiple check boxes to be checked from a group of check boxes, and a `RadioButton` control allows only one radio button to be selected from a group of mutually exclusive radio buttons.
- The `CheckBox` control can allow you to set three check states—`Checked`, `Unchecked`, and `Indeterminate`—if the `ThreeState` property is set to `true`.
- The `ComboBox` control allows you to select a value from a list of predefined values or to enter a value in the combo box. The `ListBox` control allows you to select a value from a list of values displayed.
- The `CheckedListBox` control derives from the `ListBox` control and inherits its functionality. However, a `CheckedListBox` control displays a check box along with a list of items to be checked. It allows only two selection modes: `None` (allows no selection) and `One` (allows multiple selections).

The `DomainUpDown` and `NumericUpDown` Controls

The `DomainUpDown` and `NumericUpDown` controls inherit from the `System.Windows.Forms.UpDownBase` class. You use them to select values from the (generally) ordered collection of values by pressing the control's up and down buttons. You can also enter values in these controls, unless the `ReadOnly` property is set to `true`.

The `DomainUpDown` control allows you to select from a collection of objects. When an item is selected, the object is converted to `String` and is displayed. If you want a control that displays numeric values, you instead use the `NumericUpDown` control. Table 2.14 summarizes the important members of the `DomainUpDown` class.

TABLE 2.14

IMPORTANT MEMBERS OF THE `DomainUpDown` CLASS

Member	*Type*	*Description*
`Items`	Property	Represents a collection of objects assigned to a control.
`ReadOnly`	Property	Indicates whether you can change the value in a way other than by pressing the up and down buttons.
`SelectedIndex`	Property	Specifies the index value of the selected item in the items collection.
`SelectedItem`	Property	Specifies the value of the selected item.
`SelectedItemChanged`	Event	Occurs when the `SelectedIndex` property is changed. `SelectedItemChanged` is the default event of the `DomainUpDown` class.
`Sorted`	Property	Indicates whether the items collection is sorted.
`Wrap`	Property	Indicates whether the `SelectedIndex` property resets to the first or the last item if the user continues past either end of the list.

The `NumericUpDown` control contains a single numeric value that can be increased or decreased when you click the up or down buttons of the control. You can specify the `Minimum`, `Maximum`, or `Increment` value to control the range of values in this control. Table 2.15 summarizes the important members of the `NumericUpDown` class.

TABLE 2.15

IMPORTANT MEMBERS OF THE `NumericUpDown` CLASS

Member	*Type*	*Description*
`Increment`	Property	Indicates to increase or decrease the `Value` property by the specified amount when the up or down button is clicked
`Maximum`	Property	Specifies the maximum allowed value
`Minimum`	Property	Specifies the minimum allowed value

Member	*Type*	*Description*
`ReadOnly`	Property	Indicates whether you can change the value in a way other than by pressing the up and down buttons
`ThousandsSeparator`	Property	Indicates whether the thousands separator should be used when appropriate
`Value`	Property	Specifies a value assigned to a control
`ValueChanged`	Event	Occurs when the `Value` property is changed. `ValueChanged` is the default event of the `NumericUpDown` class.

STEP BY STEP

2.12 Using `DomainUpDown` and `NumericUpDown` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_12`.

2. Add three `Label` controls, one `DomainUpDown` control, and one `NumericUpDown` control to the form and arrange them as shown in Figure 2.26.

3. Name the `DomainUpDown` control `dudColor`. Set its `Text` property to `Black`, `UpDownAlign` property to `Left`, and `Wrap` property to `true`. Select its `Items` property and click the ellipsis (…) button. In the String Collection Editor add the following values:

```
Violet
Indigo
Blue
Green
Yellow
Orange
Red
Black
White
```

4. Name the `NumericUpDown` control `nudSize`. Set its `Maximum` property to `30`, `Minimum` to `2`, `Increment` to `2` , `Value` to `12`, and `ReadOnly` to `true`.

continues

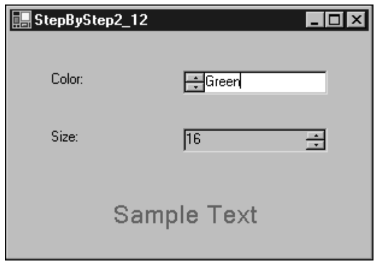


FIGURE 2.26
The `DomainUpDown` and `NumericUpDown` controls allow the user to select a value from a given list of values, by clicking up or down buttons or by directly entering a value.

continued

5. Name the `Label` placed at the bottom of the form `lblSampleText`, and then change its `Text` property to `sample Text` and its `TextAlign` property to `MiddleCenter`.

6. Attach the event handlers to the default events of the `DomainUpDown` and `NumericUpDown` controls. Add the following code to the event handlers:

```
private void dudColor_SelectedItemChanged(
    object sender, System.EventArgs e)
{
    //Typecast the object to DomainUpDown
    DomainUpDown dudColor = (DomainUpDown) sender;
    //Change color of lblsampleText to selected color
    lblSampleText.ForeColor =
       Color.FromName(dudColor.Text);
}

private void nudSize_ValueChanged(
     object sender, System.EventArgs e)
{
   //Typecast the object to NumericUpDown
   NumericUpDown nudSize = (NumericUpDown) sender;
   //Change the font of lblSampleText to selected font
   lblSampleText.Font = new Font(
     lblSampleText.Font.FontFamily,
     (float) nudSize.Value);
}
```

7. Insert the `Main()` method to launch form `StepByStep2_12`. Set the form as the startup object for the project.

8. Run the project. Click the up and down buttons of the `UpDown` controls. Their respective event handlers are fired and change the appearance of the `Text` property of the `lblSampleText` control (refer to Figure 2.26). You can enter a desired color in the `DomainUpDown` control because its `ReadOnly` property is `false`.

The `MonthCalendar` and `DateTimePicker` Controls

The `MonthCalendar` control provides a user-friendly interface to select a date or a range of dates. Table 2.16 summarizes the important members of the `MonthCalendar` class with which you should be familiar.

TABLE 2.16

IMPORTANT MEMBERS OF THE MonthCalendar CLASS

Member	*Type*	*Description*
`CalendarDimensions`	Property	Specifies the number of columns and rows of months to display.
`DateChanged`	Event	Occurs when the date that is selected in the control changes. `DateChanged` is the default event of the `MonthCalendar` class.
`DateSelected`	Event	Occurs when a date is selected in the control.
`FirstDayOfWeek`	Property	Represents the first day of the week displayed in the calendar.
`MaxDate`	Property	Specifies the latest allowable date that can be selected.
`MaxSelectionCount`	Property	Specifies the maximum number of days that can be selected.
`MinDate`	Property	Specifies the earliest allowable date that can be selected.
`SelectionEnd`	Property	Specifies the end date of the selected range of dates.
`SelectionRange`	Property	Specifies the selected range of dates.
`SelectionStart`	Property	Specifies the start date of the selected range of dates.
`ShowToday`	Property	Indicates whether today's date should be displayed in the bottom of the control.
`ShowTodayCircle`	Property	Indicates whether today's date should be circled.
`ShowWeekNumbers`	Property	Indicates whether the week numbers (1–52) should be displayed at the beginning of each row of days.
`TodayDate`	Property	Represents today's date.

The `DateTimePicker` control allows the user to select the date and time in different formats. The `Format` property determines the format in which the control displays the date and time. You can also use the `DateTimePicker` control to display a custom date/time format by setting the `Format` property to `DateTimePickerFormat.Custom` and the `CustomFormat` property to the custom format desired. Table 2.17 summarizes the important members of the `DateTimePicker` class.

TABLE 2.17

IMPORTANT MEMBERS OF THE DateTimePicker CLASS

Member	*Type*	*Description*
CustomFormat	Property	Represents the custom date/time format string.
Format	Property	Specifies the format of the date and time that are displayed in the control. The values are specified by the DateTimePickerFormat enumeration—Custom, Long (the default), Short, and Time. Long, Short, and Time display the date in the value formats set by the operating system. The Custom value lets you specify a custom format.
FormatChanged	Event	Occurs when the Format property changes.
MaxDate	Property	Specifies the latest allowable date and time to be selected.
MinDate	Property	Specifies the soonest allowable date and time to be selected.
ShowCheckBox	Property	Indicates whether the check box should be displayed to the left of the selected date.
ShowUpDown	Property	Indicates whether an UpDown control, rather than the default calendar control, should be displayed to allow the user to make selections.
Value	Property	Represents the value of the date and time selected.
ValueChanged	Event	Occurs when the Value property changes. ValueChanged is the default event of the DateTimePicker class.

STEP BY STEP

2.13 Using MonthCalendar and DateTimePicker Controls

1. Add a Windows form to existing project 316C02. Name the form StepByStep2_13.

2. Place three `Label` controls, one `MonthCalendar` control (`mcTravelDates`), one `DateTimePicker` control (`dtpLaunchDate`), and two `RadioButton` controls (`rbLongDate` and `rbShortDate`) on the form and arrange them as shown in Figure 2.27. Name the `Label` control that is placed adjacent to the `MonthCalendar` control `lblTravelDates`.

3. Switch to the code view and add the following `using` directive:

```
using System.Text;
```

4. Add an event handler for the `DateSelected` event for the `MonthCalendar` control. Add the following code to the event handler:

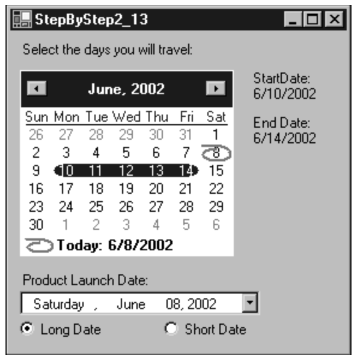
```
private void mcTravelDates_DateSelected(object sender,
    System.Windows.Forms.DateRangeEventArgs e)
{
    StringBuilder sbMessage = new StringBuilder();
    sbMessage.Append("StartDate:\n");
    sbMessage.Append(e.Start.ToShortDateString());
    sbMessage.Append("\n\nEnd Date:\n");
    sbMessage.Append(e.End.ToShortDateString());
    this.lblTravelDates.Text = sbMessage.ToString();
}
```

5. Attach the event handlers to the default events of the `RadioButton` controls. Add the following code to the event handlers:

```
private void rbLongDate_CheckedChanged(
    object sender, System.EventArgs e)
{
    if(rbLongDate.Checked)
        dtpLaunchDate.Format =
          DateTimePickerFormat.Long;
}

private void rbShortDate_CheckedChanged(
    object sender, System.EventArgs e)
{
    if(rbShortDate.Checked)
        dtpLaunchDate.Format =
          DateTimePickerFormat.Short;
}
```

6. Insert a `Main()` method to launch form `StepByStep2_13`. Set the form as the startup object for the project.

continues



StartDate:
6/10/2002

End Date:
6/14/2002

FIGURE 2.27
You can perform date selections using the `MonthCalendar` and `DateTimePicker` controls.

continued

7. Run the project. Select a range of dates from the `MonthCalendar` control and a date from the `DateTimePicker` control. The label adjacent to the `MonthCalendar` control displays the start date and end date from the range of the dates selected (refer to Figure 2.27). You can also change the format of the date shown by the `DateTimePicker` control by selecting radio buttons.

The `Timer`, `TrackBar`, and `ProgressBar` Controls

The `Timer` control is used when an event needs to be generated at user-defined intervals. Table 2.20 summarizes the important members of the `Timer` class.

EXAM TIP

`System.Windows.Forms.Timer` Versus `System.Timers.Timer` You should always use the `Timer` class from the `System.Windows.Forms` namespace in Windows Forms because this class is optimized to work with forms. The `System.Timers.Timer` class fires `Tick` events from another thread and may cause indeterminate results in applications.

TABLE 2.20

IMPORTANT MEMBERS OF THE `Timer` CLASS

Member	*Type*	*Description*
`Enabled`	Property	Indicates whether the timer is currently running.
`Interval`	Property	Represents the time, in milliseconds, between ticks of the timer.
`Start()`	Method	Starts the `Timer` control.
`Stop()`	Method	Stops the `Timer` control.
`Tick`	Event	Occurs when the timer interval elapses and the timer is enabled.

A `TrackBar` control provides an intuitive way to select a value from a given range by providing a scroll box and a scale of value. The user can slide the scroll box on the scale to point to the desired value. Table 2.21 summarizes the important members of the `TrackBar` class.

TABLE 2.21

IMPORTANT MEMBERS OF THE `TrackBar` CLASS

Member	*Type*	*Description*
`LargeChange`	Property	Indicates the number of ticks by which the `Value` property changes when the scroll box is moved a large distance.

continues

TABLE 2.21 *continued*

IMPORTANT MEMBERS OF THE TrackBar CLASS

Member	*Type*	*Description*
`Maximum`	Property	Specifies the upper bound of the `TrackBar` control's range.
`Minimum`	Property	Specifies the lower bound of the `TrackBar` control's range.
`Orientation`	Property	Represents the horizontal or vertical orientation of the control.
`Scroll`	Event	Occurs when the scroll box is moved by a keyboard or mouse action. `Scroll` is the default event for the `TrackBar` class.
`SmallChange`	Property	Indicates the number of ticks by which the `Value` property changes when the scroll box is moved a small distance.
`TickFrequency`	Property	Represents the frequency within which ticks are drawn in the control.
`TickStyle`	Property	Represents the way the control appears. The values are specified by the `TickStyle` enumeration—`Both`, `BottomRight`, `None`, and `TopLeft`.
`Value`	Property	Represents the scroll box's current position in the control.
`ValueChanged`	Property	Occurs when the `Value` property changes via the `Scroll` event or programmatically.

A `ProgressBar` control is usually displayed to indicate the status of a lengthy operation such as installing an application, copying files, or printing documents. Table 2.22 summarizes the important members of the `ProgressBar` class.

TABLE 2.22

IMPORTANT MEMBERS OF THE ProgressBar CLASS

Member	*Type*	*Description*
`Maximum`	Property	Specifies the upper bound of the progress bar's range.
`Minimum`	Property	Specifies the lower bound of the progress bar's range.
`Value`	Property	Represents the current position of the control.

Step by Step 2.15 gives an example of using the `TrackBar` and `ProgressBar` controls. The example simulates a lengthy operation with the help of a `Timer` control. You can control the speed with which the process works by using a `TrackBar` control that changes the `Interval` property of the `Timer` control to set the time at which it will generate `Tick` events.

STEP BY STEP

2.15 Using `Timer`, `TrackBar`, and `ProgressBar` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_15`.
2. Place a `Timer` control on the form. This control is added to the component tray. Name the `Timer` control `tmrTimer` and set its `Enabled` property to `true`.
3. Place four `Label` controls, one `ProgressBar` control, and one `TrackBar` control on the form and arrange them as shown in Figure 2.30.
4. Name the `ProgressBar` control `prgIndicator` and the `TrackBar` control `trkSpeed`. For the `TrackBar` control, change the `Maximum` property to `1000`, `TickFrequency` to `100`, `TickStyle` to `Top,Left`, and `Value` to `100`.
5. Name a `Label` control `lblMessage`. Change the label's `Size - Height` property to `1` and `BorderStyle` property to `Fixed3D`, to represent it as a line.
6. Double-click the `Timer` and `TrackBar` controls to attach default event handlers to their default events—`Tick` and `Scroll`, respectively. Add the following code to the event handlers:

```
private void tmrTimer_Tick(
    object sender, System.EventArgs e)
{
    if (prgIndicator.Value < prgIndicator.Maximum)
        //Increase the progress indicator
        prgIndicator.Value += 5;
    else
```

continues

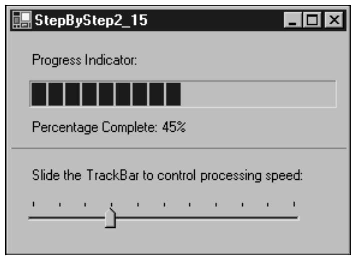


FIGURE 2.30
The `Timer` control updates the progress bar on every tick, and the `TrackBar` control controls the interval of `Tick` events for the `Timer` control.

NOTE

No Line and Shape Controls Unlike in earlier versions of Visual Studio, you won't find any `Line` or `Shape` control in Visual Studio .NET. This is because all controls in the Visual Studio .NET toolbox must be windowed. The `Line` and `Shape` controls are window-less, and hence they were removed. What can you do about it? For drawing a simple line, you can use a `Label` control in which you can set the `Height` property to `1` (or more, if you want a thicker line) and set the `BorderStyle` property. For more advanced lines and shapes, you can use powerful GDI+ classes that are available in the Windows Forms library (refer to Chapter 1 for discussion on the `System.Drawing` namespace).

continued

```
            //Reset the progress bar indicator
            prgIndicator.Value = prgIndicator.Minimum;
        lblMessage.Text = "Percentage Complete: " +
            prgIndicator.Value + "%";
    }

    private void trkSpeed_Scroll(
        object sender, System.EventArgs e)
    {
        TrackBar trkSpeed = (TrackBar) sender;
        if (trkSpeed.Value >= 1)
            //Set timer value based on user's selection
            tmrTimer.Interval = trkSpeed.Value;
    }
```

7. Insert the `Main()` method to launch form `StepByStep2_15`. Set the form as the startup object for the project.

8. Run the project. Slide the `TrackBar` control, and the progress bar progresses at different speeds, depending on the time interval set by the `TrackBar` control (refer to Figure 2.30).

The `HScrollBar` and `VScrollBar` Controls

The `HScrollBar` and `VScrollBar` controls display horizontal and vertical scrollbars, respectively. The `HScrollBar` and `VScrollBar` classes inherit their properties and other members from the `ScrollBar` class, which provides the basic scrolling functionality.

Usually, controls such as `Panel`, `TextBox`, and `ComboBox` include their own scrollbars. But some controls, such as `PictureBox`, do not have built-in scrollbars. You can use `HScrollBar` and `VScrollBar` to associate scrollbars with such controls. Table 2.23 summarizes the important members of `ScrollBar` class, from which the `HScrollBar` and `VScrollBar` controls inherit.

TABLE 2.23

IMPORTANT MEMBERS OF THE ScrollBar CLASS

Member	*Type*	*Description*
`LargeChange`	Property	Indicates the number by which the `Value` property changes when the scroll box is moved a large distance.
`Maximum`	Property	Specifies the upper bound of the scrollbar's range.
`Minimum`	Property	Specifies the lower bound of the scrollbar's range.
`Scroll`	Event	Occurs when the scroll box is moved by a keyboard or mouse action. `Scroll` is the default event for the `ScrollBar` class.
`SmallChange`	Property	Indicates the number by which the `Value` property changes when the scroll box is moved a small distance.
`Value`	Property	Represents the current position of the control.
`ValueChanged`	Event	Occurs when the `Value` property changes either via the `Scroll` event or programmatically.

STEP BY STEP

2.16 Using `HScrollBar` and `VScrollBar` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_16`.
2. Place a `PictureBox` control, an `HScrollBar` control, and a `VScrollBar` control on the form. Name the `PictureBox` control `pbImage` and set the `SizeMode` property to `AutoSize`. Select the `Image` property and click the ellipsis (…) button. This causes an Open File dialog box to appear. Select the image you want to have uploaded in the form.
3. Name the `HScrollBar` control `hScroll` and set its `Dock` property to `Bottom`. Change the `Name` property of the `VScrollBar` control to `vScroll` and set its `Dock` property to `Right`.

continues

continued

4. Double-click the `HScrollBar` and `VScrollBar` controls to attach event handlers to their default `Scroll` events. Add the following code to the event handlers:

```
private void vScroll_Scroll(object sender,
    System.Windows.Forms.ScrollEventArgs e)
{
    //Scroll the image vertically
    pbImage.Top = vScroll.Bottom - pbImage.Height -
       (int)getVScrollAdjustment();
}
private float getVScrollAdjustment()
{
    //Calculate vertical scroll bar changes
    float vPos =
      (float)(vScroll.Value - vScroll.Minimum);
    float vDiff =
       (float)(vScroll.Height - pbImage.Height);
    float vTicks =
       (float)(vScroll.Maximum - vScroll.Minimum);
    return (vDiff/vTicks)*vPos;
}

private void hScroll_Scroll(object sender,
    System.Windows.Forms.ScrollEventArgs e)
{
    //Scroll the image horizontally
    pbImage.Left = hScroll.Right - pbImage.Width -
       (int)getHScrollAdjustment();
}
private float getHScrollAdjustment()
{
    //Calculate horizontal scrollbar changes
    float hPos =
        (float)(hScroll.Value - hScroll.Minimum);
    float hDiff =
        (float)(hScroll.Width - pbImage.Width);
    float hTicks =
        (float)(hScroll.Maximum - hScroll.Minimum);

    return (hDiff/hTicks)*hPos;
}
```

5. Insert the `Main()` method to launch form `StepByStep2_16`. Set the form as the startup object for the project.

6. Run the project. The form displays the image, and the scrollbars that are on the right and bottom can be used to scroll through the image if the image size exceeds the allotted space, as in Figure 2.31.

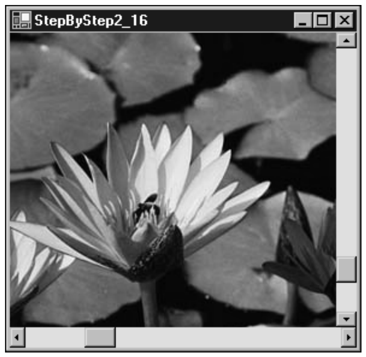


FIGURE 2.31
The `HScrollBar` and `VScrollBar` classes allow you to implement scrolling functionality in an application.

The TabControl Control

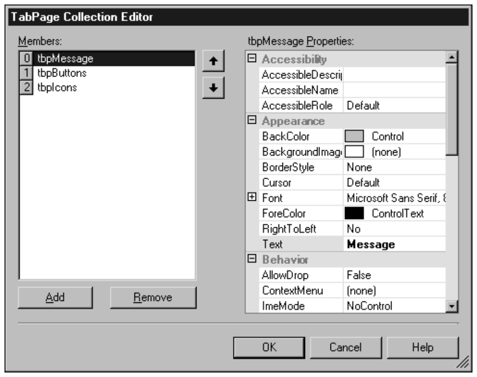
The TabControl control displays a collection of tabbed pages. Each tabbed page can contain its own controls. The TabControl control can be useful in organizing large number of controls because when the controls are organized into various tabbed pages they occupies less space on the form. Tabbed pages appear mostly in wizards and IDEs. For example, the Visual Studio .NET IDE displays all the open files in tabbed pages. Table 2.24 summarizes the important members of the TabControl class.

TABLE 2.24

IMPORTANT MEMBERS OF THE TabControl CLASS

Member	*Type*	*Description*
Alignment	Property	Represents the area where the tabs will be aligned—Bottom, Left, Right, or Top (the default).
ImageList	Property	Represents the ImageList control from which images are displayed on tabs.
MultiLine	Property	Indicates whether tabs can be displayed in multiple rows.
SelectedIndex	Property	Represents the index of the selected tabbed page.
SelectedIndexChanged	Event	Occurs, when the selected index changes. SelectedIndexChanged is the default event of the TabControl class.
SelectedTab	Property	Specifies the selected tabbed page.
TabCount	Property	Specifies a count of the tabs in a control.
TabPages	Property	Specifies a collection of tabbed pages in a control.

Step by Step 2.17 displays a TabControl control to build a message box by accepting the values for the caption and the message in one tab page, the message box buttons in the second tab page, and the icon to be displayed in the third tab page.



Members:

0 tbpMessage
1 tbpButtons
2 tbpIcons

tbpMessage Properties:

Accessibility	
AccessibleDescri	
AccessibleName	
AccessibleRole	Default
Appearance	
BackColor	Control
BackgroundImag	(none)
BorderStyle	None
Cursor	Default
Font	Microsoft Sans Serif, 8
ForeColor	ControlText
RightToLeft	No
Text	**Message**
Behavior	
AllowDrop	False
ContextMenu	(none)
ImeMode	NoControl

FIGURE 2.32
You can use the TabPage Collection Editor to add tab pages to a `TabControl` control.

STEP BY STEP

2.17 Using `TabControl` Controls

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_17`.

2. Place a `TabControl` control (`tabDemo`) on the form. Select the `TabPages` property and click the ellipsis (…) button. This invokes the TabPage Collection Editor. Click the Add button and a `TabPage` control is added to the collection, with its index. Name the `TabPage` control `tbpMessage` and change its `Text` property to `Message`. Add two more `TabPage` controls: `tbpButtons` with `Text` property `Buttons` and `tbpIcons` with `Text` property `Icons`. Figure 2.32 shows the TabPage Collection editor after you add the tab pages. Click OK to close the TabPage Collection Editor.

3. Place two `Label` controls and two `TextBox` controls (`txtMessage` and `txtCaption`) on the Message tab page, five `RadioButton` controls (`rbOK`, `rbOKCCancel`, `rbRetryCancel`, `rbYesNo`, and `rbYesNoCancel`) on the Buttons tab page, and five `RadioButton` controls (`rbError`, `rbInformation`, `rbNone`, `rbQuestion`, and `rbWarning`) on the Icons tab page. Place a `GroupBox` control on the form and four `RadioButton` controls (`rbLeft`, `rbRight`, `rbTop`, and `rbBottom`) inside it. Place a `Button` control (`btnShow`) on the form. Arrange all the controls and set their `Text` properties as shown in Figure 2.33.

4. Switch to the code view and add the following code before the constructor definition:

```
private MessageBoxButtons mbbButtons;
private MessageBoxIcon mbiIcon;
```

5. Add the following code in the constructor:

```
public StepByStep2_17()
{
    //
    // Required for Windows Forms Designer support
    //
    InitializeComponent();
    //Initial setting for MessageBox button
    mbbButtons = MessageBoxButtons.OK;
    //Initial setting for MessageBox icon
    mbiIcon = MessageBoxIcon.Information;
}
```

6. Add the following code in the code view:

```
//This event handler is used by the RadioButtons
//that control TabControl's alignment
private void rbAlign_CheckedChanged(
    object sender, System.EventArgs e)
{
    //Typecast sender object to a RadioButton
    RadioButton rbAlign = (RadioButton) sender;

    //Only if the radio button was checked
    if(rbAlign.Checked)
    {
        //Set the alignment of TabControl based
        //on which RadioButton was checked.
        if (rbAlign == rbLeft)
            tabDemo.Alignment = TabAlignment.Left;
        else if (rbAlign == rbRight)
            tabDemo.Alignment = TabAlignment.Right;
        else if (rbAlign == rbBottom)
            tabDemo.Alignment = TabAlignment.Bottom;
        else
            tabDemo.Alignment = TabAlignment.Top;
    }
}

private void rbButtons_CheckedChanged(
    object sender, System.EventArgs e)
{
    //find the RadioButton that was checked
    //from the Buttons tab and create a
    //MessageBoxButtons objects corresponding to it.
    if(sender == rbOKCancel)
        mbbButtons = MessageBoxButtons.OKCancel;
    else if (sender == rbRetryCancel)
        mbbButtons = MessageBoxButtons.RetryCancel;
    else if (sender == rbYesNo)
        mbbButtons = MessageBoxButtons.YesNo;
    else if (sender == rbYesNoCancel)
        mbbButtons = MessageBoxButtons.YesNoCancel;
    else
        mbbButtons = MessageBoxButtons.OK;
}

private void rbIcon_CheckedChanged(
    object sender, System.EventArgs e)
{
```

continues

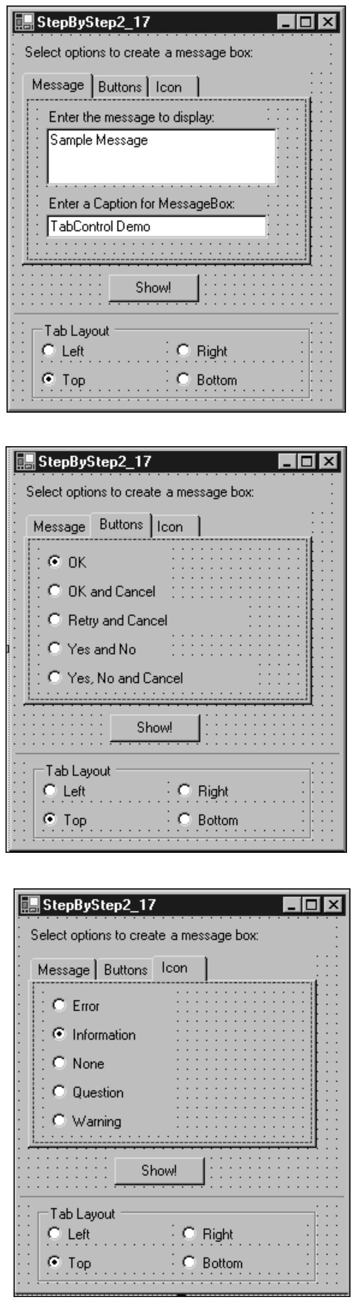


FIGURE 2.33
You can organize various controls on each tab page of a `TabControl` control.

continued

```
    //find the RadioButton that was checked
    //from the Icon tab and create a
    //MessageBoxIcon objects corresponding to it.
    if(sender == rbError)
        mbiIcon = MessageBoxIcon.Error;
    else if (sender == rbWarning)
        mbiIcon = MessageBoxIcon.Warning;
    else if (sender == rbNone)
        mbiIcon = MessageBoxIcon.None;
    else if (sender == rbQuestion)
        mbiIcon = MessageBoxIcon.Question;
    else
        mbiIcon = MessageBoxIcon.Information;
}
```

7. Select all the `RadioButton` controls in the Buttons tab page. Invoke the Properties window and click the Events icon. Select the `CheckedChanged` event and select the `rbButtons_CheckChanged` event handler from the list. Repeat the same steps for the `Icon` tab page by selecting all the `RadioButton` controls in the Icon tab page, and select the `rbIcon_CheckedChanged` event handler. Select all the `RadioButton` controls in the group box and then again select the `CheckedChanged` event and select the `rbAlign_CheckChanged` event handler from the list of event handlers.

8. Double-click the `Button` control and add the following code to handle the `Click` event of the `Button` control:

```
private void btnShow_Click(
    object sender, System.EventArgs e)
{
    MessageBox.Show(txtMessage.Text,
       txtCaption.Text, mbbButtons, mbiIcon);
}
```

9. Insert the `Main()` method to launch form `StepByStep2_17`. Set the form as the startup object for the project.

10. Run the project. Enter the message and caption of the message in the Message tab page, select the desired button from the Buttons tab page, and select the desired icon from the Icons tab page. Click the Show! button. A message box appears, with the desired message, caption, button, and icon. You can align tabs to different directions, depending on the alignment side selected from the Tab Layout group box.

R E V I E W   B R E A K

- `DomainUpDown` and `NumericUpDown` allow you to select from a list of defined values by pressing up and down buttons. You can also directly enter values in these controls unless their `ReadOnly` properties are set to `true`.
- The `DateTimePicker` control allows you to select a date and time, and the `MonthCalendar` control allows you to select a date or range of dates. The `SelectionStart`, `SelectionEnd`, and `SelectionRange` properties of the `MonthCalendar` control return the start date, end date, and range of dates selected.
- `ScrollBar` controls can be associated with controls to provide scrolling functionality.
- The `TabControl` control provides a user interface that can be used to save space as well as organize large numbers of controls. You often see `TabControl` controls used in `wizards`.

CREATING MENUS AND MENU ITEMS

Add controls to a Windows form.

- **Create menus and menu items.**

Windows applications use menus to provide organized collections of commands that can be performed by the user. Menus can inform the user of an application's capabilities as well as its limitations. If a program has properly organized menus, users can easily find common commands as well as less familiar features. Users can also learn shortcut keys from a well-designed menu structure.

Because menus have so many benefits to the user, you should be well versed in their creation and the function they can provide. The following sections discuss the menu-related classes `MainMenu`, `MenuItem`, and `ContextMenu`. They also discuss two controls, the `StatusBar` and `ToolBar` controls, that are often used with menus.

All three of the menu-related classes `MainMenu`, `MenuItem`, and `ContextMenu` derive from the `Menu` class, which provides common functionality to these classes. The `MainMenu` class is used to create an application's top-level menu and the `ContextMenu` class is used to create a shortcut menu that appears when the user right-clicks a control. Both `MainMenu` and `ContextMenu` have `MenuItems` properties. `MenuItems` is a collection of `MenuItem` objects, each representing an individual menu item. It is interesting to note that `MenuItem` itself also has a menu items collection that can be used to store submenus, thereby creating a hierarchical menu structure. Table 2.25 summarizes the important members of the `MenuItem` class with which you should be familiar.

TABLE 2.25

IMPORTANT MEMBERS OF THE `MenuItem` CLASS

Member	*Type*	*Description*
`Checked`	Property	Indicates whether a checkmark or a radio button should appear near the menu item.
`Click`	Event	Occurs when the user selects the menu item.
`DrawItem`	Event	Occurs when a request is made to draw an owner-drawn menu item. `DrawItem` occurs only when the `OwnerDraw` property is set to `true`.
`Enabled`	Property	Indicates whether the menu item is enabled.
`MenuItems`	Property	Specifies the collection of `MenuItem` objects associated with the menu. This property can be used to create hierarchical submenus.
`OwnerDraw`	Property	Indicates whether you can provide your own custom code to draw a menu item instead of letting Windows handle it in a standard way.
`Parent`	Property	The parent with which the menu item is associated. You must specify a parent for a `MenuItem` object; otherwise, it is not displayed.
`PerformClick`	Method	Generates the `Click` event for the menu item as if the user had clicked it.

Member	*Type*	*Description*
`Popup`	Event	Occurs just before a submenu corresponding to the menu item is displayed. This event handler is generally used to add, remove, enable, disable, check, or uncheck menu items, depending on the state of an application just before these menu items are displayed.
`RadioCheck`	Property	Indicates whether the menu item should display a radio button instead of a checkmark when its `Checked` property is `true`.
`Select`	Event	Occurs when the user selects a menu item by navigating to it.
`Shortcut`	Property	Specifies the shortcut key combination associated with the menu item.
`Text`	Property	Specifies the caption of the menu item.

The `MainMenu` Control

The `MainMenu` control is a container control for a form's main menu, which is displayed just below its title bar. Visual Studio .NET provides an easy-to-use menu designer that helps you quickly design a main menu for a form.

A Windows form can have only one `MainMenu` object associated with it, and it is identified by the `Menu` property of the `Form` object. After you create a menu, you should be sure to set the `Menu` property of the form to the name of the menu you want to display on it.

The most important member of the `MainMenu` class is `MenuItems`, which is a collection of `MenuItem` objects.

In Step by Step 2.18 you create a simple word processing program that provides little functionality but gives a good overview of how menus are used in a Windows application. Later in this chapter you use the information from Step by Step 2.18 along with extra features such as status bars and toolbars.

You will learn the following from Step by Step 2.18:

- How to add menus and submenus to a form
- How to use `Checked` and `RadioButton` menu items, and how to select and deselect them based on user actions

- How to associate hotkeys and shortcuts with menu items
- How to write event handlers for performing actions when a user selects a menu item

STEP BY STEP

2.18 Creating a Main Menu for a Form

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_18`.
2. Place a panel on the form and set the `Dock` property to `Fill` and `AutoScroll` to `true`.
3. Add a `RichTextBox` control to the panel. Name it `rbText` and change its `Dock` property to `Fill`.
4. From the toolbox drag a `MainMenu` control onto the form. Change the control's `Name` property to `mnuMainMenu` and change the form's `Menu` property to `mnuMainMenu`.
5. Using the menu designer (see Figure 2.34), add a top-level menu item. Set its `Text` as `&File` and name it `mnuFile`. Add the menu items listed in Table 2.26 to it.

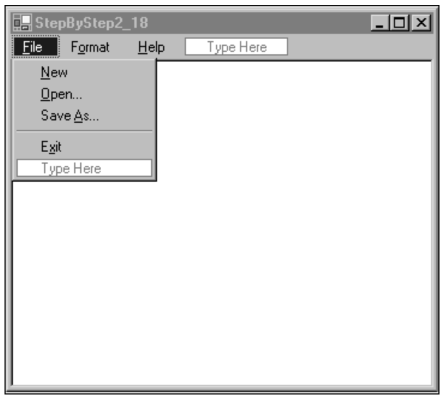


FIGURE 2.34
The Menu Designer allows you to create structured menus within the Windows Forms Designer.

TABLE 2.26
FILE MENU ITEMS

Text	*Control Name*	*Shortcut*
`&New`	`mnuFileNew`	`Ctrl+N`
`&Open...`	`mnuFileOpen`	`Ctrl+O`
`Save &As...`	`mnuFileSaveAs`	`Ctrl+S`
`E&xit`	`mnuFileExit`	None

6. Create the second top-level menu item, with its `Text` property as `F&ormat` and `Name` property as `mnuFormat`. Add the menu items listed in Table 2.27 to it.

TABLE 2.27

FORMAT MENU ITEMS

Text	*Control Name*	*Shortcut*
`&Color`	`mnuFormatColor`	None
`&Font`	`mnuFormatFont`	None

7. Set up the Format, Color menu according to the items listed in Table 2.28.

TABLE 2.28

FORMAT, COLOR MENU ITEMS

Text	*Control Name*	*Shortcut*	*`RadioCheck`* *Setting*
`&All Colors...`	`mnuFormatAllColors`	None	`false`
`&Black`	`mnuFormatColorBlack`	`Ctrl+Shift+B`	`true`
`Bl&ue`	`mnuFormatColorBlue`	`Ctrl+Shift+U`	`true`
`&Green`	`mnuFormatColorGreen`	`Ctrl+Shift+G`	`true`
`&Red`	`mnuFormatColorRed`	`Ctrl+Shift+R`	`true`

8. Set up the Format, Font menu according to the items listed in Table 2.29.

TABLE 2.29

FORMAT, FONT MENU ITEMS

Text	*Control Name*	*Shortcut*
`&All Fonts...`	`mnuFormatFontAllFonts`	None
`&Bold`	`mnuFormatFontBold`	`Ctrl+B`
`&Italic`	`mnuFormatFontItalic`	`Ctrl+I`
`&Underline`	`mnuFormatFontUnderline`	`Ctrl+U`

continues

continued

9. Name the third top-level menu `mnuHelp` and set its `Text` property to `&Help`. This menu has only one menu item in it: `mnuHelpAbout`. Set the `Text` property of `mnuHelpAbout` to `&About`.

10. In the menu designer, right-click the Exit menu and select Insert Separator from the context menu. Insert separators after the All Colors… and All Fonts… menu items.

11. From the toolbox drop four dialog box components: `OpenFileDialog`, `SaveFileDialog`, `FontDialog`, and `ColorDialog`. Change their `Name` properties to `dlgOpenFile`, `dlgSaveFile`, `dlgFont`, and `dlgColor`, respectively.

12. Select File, New. In the Properties window double-click the menu item's `Click` event to add an event handler to it. Add the following code to the event handler:

```
private void mnuFileNew_Click(
    object sender, System.EventArgs e)
{
    rtbText.Clear();
}
```

13. Add the following code to the `Click` event handler of the File, Open menu item:

```
private void mnuFileOpen_Click(
    object sender, System.EventArgs e)
{
    //Allow to select only *.rtf files
    dlgOpenFile.Filter =
       "Rich Text Files (*.rtf)|*.rtf";
    if(dlgOpenFile.ShowDialog() == DialogResult.OK)
    {
        //Load the file contents in the RichTextBox
        rtbText.LoadFile(dlgOpenFile.FileName,
           RichTextBoxStreamType.RichText);
    }
}
```

14. Double-click the File, Save As… menu item to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void mnuFileSaveAs_Click(
    object sender, System.EventArgs e)
{
    //Default choice to save file is *.rtf
    //but user can select
    //All Files to save with other extension
    dlgSaveFile.Filter =
  "Rich Text Files (*.rtf)|*.rtf|All Files (*.*)|*.*";
    if(dlgSaveFile.ShowDialog() == DialogResult.OK)
    {
        //Save the RichText content to a file
        rtbText.SaveFile(dlgSaveFile.FileName,
            RichTextBoxStreamType.RichText);
    }
}
```

15. Double-click the File, Exit menu item to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void mnuFileExit_Click(
     object sender, System.EventArgs e)
{
    //close the form
    this.Close();
}
```

16. Double-click the Format, Color, All Colors… menu item to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void mnuFormatColorAllColors_Click(
    object sender, System.EventArgs e)
{
    if(dlgColor.ShowDialog() == DialogResult.OK)
    {
        //Change the color of selected text
        //If no text selected, change the active color
        rtbText.SelectionColor = dlgColor.Color;
    }
}
```

17. Insert the following event handler in the code and associate it with the `Click` event of the Format, Color, Black; Format, Color, Blue; Format, Color, Green; and Format, Color, Red menu items:

```
private void mnuFormatColorItem_Click(
    object sender, System.EventArgs e)
{
    MenuItem mnuItem = (MenuItem) sender;
    //Get color name, before that also get rid of
    //the '&' character in Control's Text
    rtbText.SelectionColor = Color.FromName(
        mnuItem.Text.Replace("&", ""));
```

continues

continued

```
    //uncheck all menu items inside color menu
    foreach(MenuItem m in mnuItem.Parent.MenuItems)
        m.Checked = false;
    //just check the clicked menu
    mnuItem.Checked = true;
}
```

18. Insert the following event handler in the code and associate it with the `Popup` event of Format, Color menu item:

```
private void mnuFormatColor_Popup(
    object sender, System.EventArgs e)
{
    MenuItem mnuItem = (MenuItem) sender;

    //for all menu items inside color menu
    foreach(MenuItem m in mnuItem.MenuItems)
    {
        if (m.Text.Replace("&", "") ==
            rtbText.SelectionColor.Name)
            //If it is the selected color, check it
            m.Checked = true;
        else
            //otherwise uncheck it
            m.Checked = false;
    }
}
```

19. Double-click the Format, Font, All Fonts… menu item to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void mnuFormatFontAllFonts_Click(
    object sender, System.EventArgs e)
{
    if(dlgFont.ShowDialog() == DialogResult.OK)
    {
        //Change the font of selected text
        //If no text selected, change the active font
        rtbText.SelectionFont = dlgFont.Font;
    }
}
```

20. Insert the following event handler in the code and associate it with the `Click` event of the Format, Font, Bold; Format, Font, Italic; and Format, Font, Underline menu items:

```
private void mnuFormatFontItem_Click(
    object sender, System.EventArgs e)
{
    MenuItem mnuItem = (MenuItem) sender;
    mnuItem.Checked = !mnuItem.Checked;
    FontStyle fsStyle;
    //Set the FontStyle selected by user in fsStyle
    switch (mnuItem.Text.Replace("&",""))
    {
        case "Bold":
            fsStyle = FontStyle.Bold;
            break;
        case "Italic":
            fsStyle = FontStyle.Italic;
            break;
        case "Underline":
            fsStyle = FontStyle.Underline;
            break;
        default:
            fsStyle = FontStyle.Regular;
            break;
    }
    //Create a font object, toggle the FontStyle
    //and set the new font on selection
    rtbText.SelectionFont = new Font(
        rtbText.SelectionFont.FontFamily,
        rtbText.SelectionFont.Size,
        rtbText.SelectionFont.Style^fsStyle);
}
```

21. Insert the following event handler in the code and associate it with the `Popup` event of the Format, Font menu item:

```
private void mnuFormatFont_Popup(
    object sender, System.EventArgs e)
{
    //Set the check boxes on format menu to reflect
    //users selection of Font
    mnuFormatFontBold.Checked =
        rtbText.SelectionFont.Bold;
    mnuFormatFontItalic.Checked =
        rtbText.SelectionFont.Italic;
    mnuFormatFontUnderline.Checked =
        rtbText.SelectionFont.Underline;
}
```

22. Double-click the Help, About menu item to attach a `Click` event handler to it. Add the following code to the event handler:

```
private void mnuHelpAbout_Click(
    object sender, System.EventArgs e)
{
    Form frm = new frmAbout();
    //Display an About dialog box.
    frm.ShowDialog(this);
}
```

continues

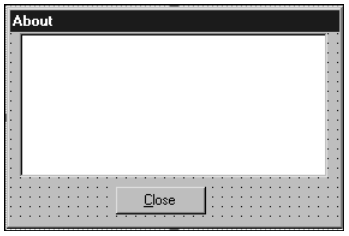


FIGURE 2.35
The About dialog box uses a `RichTextBox` control to display the contents of an RTF file.

continued

23. Add a new Windows form, named `frmAbout`, to the project. Change its `ControlBox` property to `false`, `FormBorderStyle` to `FixedDialog`, `ShowInTaskbar` to `false`, and `Text` to `About`.

24. Place a `RichTextBox` control and a `Button` control on the form and name them `rtbText` and `btnClose`, respectively. Arrange the controls as shown in Figure 2.35.

25. Add the following code to the form's `Load` event handler:

```
private void frmAbout_Load(
    object sender, System.EventArgs e)
{
    if (File.Exists("About.rtf"))
        //Load content from a file
        rtbText.LoadFile("About.rtf");
    else
        //When file not available,
        //just link to a Web site
        rtbText.Text =
         "Please visit http://www.microsoft.com/net" +
         " to learn how this product was developed";
}
```

26. Add the following code to the `LinkClicked` event handler of `rtbText`:

```
private void rtbText_LinkClicked(object sender,
     System.Windows.Forms.LinkClickedEventArgs e)
{
    //Start internet explorer to open the link
    System.Diagnostics.Process.Start(
        "IExplore", e.LinkText);
}
```

27. Add the following code to the `Close` event handler of `btnClose`:

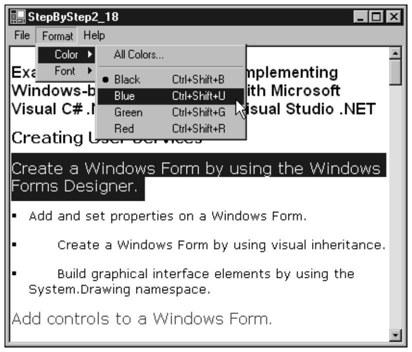
```
private void btnClose_Click(
    object sender, System.EventArgs e)
{
    //get rid of this form.
    this.Close();
}
```

28. Insert the `Main()` method to launch the form `StepByStep2_18`. Set the form as the startup object for the project.

29. Run the project. Open or create a rich text file and use the Format menu to format the text. Also use hotkeys and shortcut keys to select menu items (see Figure 2.36).

EXAM TIP

The `WordWrap` Property and Scrollbars When the `WordWrap` property of a `RichTextBox` control is `true` (the default value), the horizontal scrollbars are not displayed, regardless of the setting of the `ScrollBars` property.



Exa mplementing
Windows-b ith Microsoft
Visual C# .N isual Studio .NET
Creating U

Create a Windows Form by using the Windows Forms Designer.

- Add and set properties on a Windows Form.
- Create a Windows Form by using visual inheritance.
- Build graphical interface elements by using the System.Drawing namespace.

Add controls to a Windows Form.

FIGURE 2.36
You can use either the mouse or hotkeys or shortcut keys to access menu commands.

In Step by Step 2.18, you place a panel on the form and place the RichTextBox control on the panel. Actually, this example works fine even without using the panel. However, the panel is helpful if you want to use other controls, such as StatusBar and Toolbar, in the application. By using a Panel control, it is easy to divide a form's real estate and avoid overlap problems.

The ContextMenu Control

A context menu is typically used to provide users with a small context-sensitive menu based on the application's current state and the user's current selection. A context menu is invoked when the user right-clicks a control.

You use the ContextMenu class to create a context menu. A context menu is simpler than a main menu because it has only one top-level menu that displays all the menu items. In addition, you can have submenus and other functionality, such as hide, show, enable, disable, check, and uncheck available in a context menu.

A context menu is associated with a control or a form, so you can have several ContextMenu objects in an application, each working in a different context. You must associate a ContextMenu object with a control by assigning it to the control's ContextMenu property.

The process of creating a context menu is similar to the process of creating a main menu. Step by Step 2.19 extends the application created in Step by Step 2.18 by adding a context menu for basic editing operations such as cut, copy, and paste. You will learn the following in Step by Step 2.19:

- How to create a context menu and its items
- How to associate a context menu with a control
- How to enable and disable menu items based on application state
- How to work with the Clipboard class

EXAM TIP

The RadioCheck Property Setting the RadioCheck property to true does not implicitly set mutual exclusion for menu items; you are still able to check several of them at the same time. You have to set mutual exclusion programmatically. The Popup event is an appropriate place to check a menu item and uncheck all other menu items in that group.

NOTE

Docking Controls in a Scrollable Container When docking controls within a scrollable control such as a form, you should add a child scrollable control such as the Panel control to contain any other controls, such as a RichTextBox control, that might require scrolling. You should also set the Dock property to DockStyle.Fill and the AutoScroll property to true for the child panel control. The AutoScroll property of the parent scrollable control such as a form should be set to false.

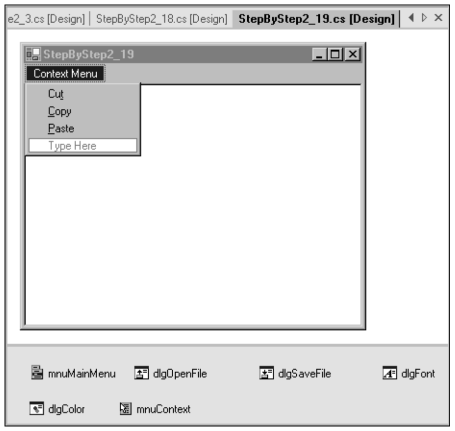


FIGURE 2.37
The Menu Designer allows you to create context menus within the Windows Forms Designer.

STEP BY STEP

2.19 Creating a Context Menu for a Form

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_19`.
2. Follow steps 2 to 27 from Step by Step 2.18.
3. From the toolbox, drag and drop a `ContextMenu` control onto the form. It is added to the component tray. Select its properties and change its name to `mnuContextMenu`.
4. Change the `ContextMenu` property of the `rtbText` object to `mnuContextMenu`.
5. Select the `ContextMenu` control. Notice that the context menu appears in place of the main menu. Create menu items in the context menu as shown in Figure 2.37, and define their properties as shown in Table 2.30.

TABLE 2.30

CONTEXT MENU ITEMS

Text	*Control Name*	*Shortcut*
`Cu&t`	`mnuContextCut`	Ctrl+X
`&Copy`	`mnuContextCopy`	Ctrl+C
`&Paste`	`mnuContextpaste`	Ctrl+V

6. Double-click the Cut menu item to add an event handler to it, and then add the following code to it:

```
private void mnuContextCut_Click(
    object sender, System.EventArgs e)
{
    //Set the clipboard with current selection
    Clipboard.SetDataObject(rtbText.SelectedRtf,true);
    //Delete the current selection
    rtbText.SelectedRtf = "" ;
}
```

7. Add the following event handler for the `Click` event handler of the Copy menu item:

```
private void mnuContextCopy_Click(
    object sender, System.EventArgs e)
{
    //Set the clipboard with current selection
    Clipboard.SetDataObject(rtbText.SelectedRtf,true);
}
```

8. Add the following code to the `Click` event handler of the Paste menu item:

```
private void mnuContextPaste_Click(
    object sender, System.EventArgs e)
{
    //DataObject provides format-independent data
    //transfer mechanism
    //Get data from clipboard and store
    //it in a DataObject object
    DataObject doClipboard =
       (DataObject)Clipboard.GetDataObject();
    //only if clipboard had any data
    if (doClipboard.GetDataPresent(DataFormats.Text))
    {
        //get the string data from DataObject object
        string text =
        (string)doClipboard.GetData(DataFormats.Text);
        if (!text.Equals(""))
            //If there was some text to paste
            //paste it in RTF format
            rtbText.SelectedRtf  = text;
    }
}
```

9. Double-click the `mnuContextMenu` object in the component tray. This adds an event handler for its `Popup` event, to which you should add the following code:

```
private void mnuContext_Popup(
     object sender, System.EventArgs e)
{
    //Initially disable all menu items
    mnuContextCut.Enabled = false;
    mnuContextCopy.Enabled = false;
    mnuContextPaste.Enabled = false;
    //If there was any selected text
    if (!rtbText.SelectedText.Equals(""))
    {
        //enable the cut and copy menu items
        mnuContextCut.Enabled = true;
        mnuContextCopy.Enabled = true;
    }
```

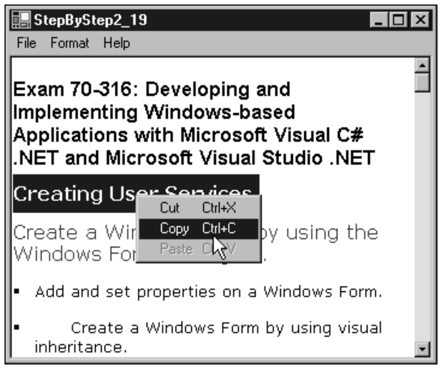
continues

continued

```
        DataObject doClipboard =
            (DataObject)Clipboard.GetDataObject();
        //if there is text data on clipboard,
        //enable the Paste menu item
        if (doClipboard.GetDataPresent(DataFormats.Text))
            mnuContextPaste.Enabled = true;
    }
```

10. Insert the `Main()` method to launch form `StepByStep2_19`. Set the form as the startup object for the project.

11. Run the project. Open or create an RTF file, select some text, and right-click and copy it to the Clipboard (see Figure 2.38). Paste it at a different location. You should also see that the Cut, Copy, and Paste menu items are enabled and disabled depending on the context. For example, if there is no text selected, the Cut and Copy menu items are disabled.



Exam 70-316: Developing and Implementing Windows-based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET

Creating User Services

Create a Wir... by using the Windows For...

- Add and set properties on a Windows Form.
- Create a Windows Form by using visual inheritance.

FIGURE 2.38
Menu options in the context menu are enabled or disabled based on the current context.

Although the Clipboard operations implemented here would work fine within this application, they might fail if you are pasting different types of data from some other application. This is because you have not implemented error handling here. Error handling is covered in detail in Chapter 3, "Error Handling for the User Interface."

GUIDED PRACTICE EXERCISE 2.3

The Edit main menu is one of the most common features of all Windows-based applications. Typically, the Edit menu contains commands such as Cut, Copy, Paste, Undo, and Redo.

In this exercise, you extend the application created in Step by Step 2.19 by adding an Edit menu as a top-level menu in the form's main menu. In addition to Cut, Copy, and Paste, you also need to implement Undo and Redo menu items that allow you to undo or redo the changes made to the `RichTextBox` control. You should try working through this problem on your own first reusing as much code as you can from Step by Step 2.19. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Add a new form to your Visual C# .NET project. Name the form `GuidedPracticeExercise2_3.cs`.
2. Follow steps 2 through 9 from Step by Step 2.19.
3. Select `mnuContextMenu` from the component tray and then in the context menu, select all menu items by pressing the Ctrl key while selecting items. Right-click and then select Copy to copy these menu items to the Clipboard. Select `mnuMainMenu`, right-click the top-level Format menu, and select Insert New from the shortcut menu. This creates a new menu item just before Format Menu. Change its `Text` property to `&Edit` and name the control `mnuEdit`. Right-click in the menu list of this newly created menu, and select Paste from the shortcut menu. All the context menu items are copied here.
4. Change the `Name` properties of the menu items Cut, Copy, and Paste to `mnuEditCut`, `mnuEditCopy`, and `mnuEditPaste`, respectively. Also change their `Shortcut` properties to `CtrlX`, `CtrlC`, and `CtrlV`, respectively.
5. Select the `mnuEditCut` menu item and choose `mnuContextCut_Click` as its `Click` event handler. Similarly, choose `mnuContextCopy_Click` and `mnuContextPaste_Click` as the event handlers for the `mnuEditCopy` and `mnuEditPaste` menus, respectively.
6. Insert another menu item just before the Edit, Cut menu item. Change its `Text` property to `&Undo`, change its `Name` property to `mnuEditUndo`, and change its `Shortcut` property to `CtrlZ`. Similarly, add another menu item for Redo and set `Text` as `&Redo`, `Name` as `mnuEditRedo`, and `Shortcut` as `CtrlY`. Insert a separator bar between Undo, Redo, and the other menu items in the Edit menu.
7. Create event handlers for the `Click` event of the `mnuEditUndo` and `mnuEditRedo` menu items. Modify the code as shown here:

```
private void mnuEditUndo_Click(
    object sender, System.EventArgs e)
{
    //Undo the last edit operation
    rtbText.Undo();
}

private void mnuEditRedo_Click(
    object sender, System.EventArgs e)
```

continues

continued

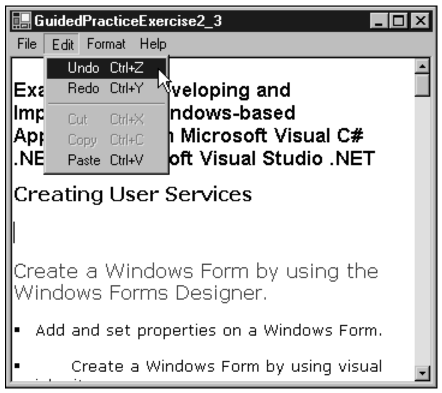
```
{
    //Redo the last operation that was undone
    rtbText.Redo();
}
```

8. Select `mnuEdit` and add the following code to its `Popup` event handler:

```
private void mnuEdit_Popup(
    object sender, System.EventArgs e)
{
    //Initially disable all menu items
    mnuEditUndo.Enabled = false;
    mnuEditRedo.Enabled = false;
    mnuEditCut.Enabled = false;
    mnuEditCopy.Enabled = false;
    mnuEditPaste.Enabled = false;

    //If there was any selected text
    if (!rtbText.SelectedText.Equals(""))
    {
        //enable the cut and copy menu items
        mnuEditCut.Enabled = true;
        mnuEditCopy.Enabled = true;
    }
    DataObject doClipboard =
        (DataObject)Clipboard.GetDataObject();
    //if there is text data on clipboard,
    //enable the Paste menu item
    if (doClipboard.GetDataPresent(DataFormats.Text))
        mnuEditPaste.Enabled = true;
    //Check if Undo is possible
    if (rtbText.CanUndo)
        mnuEditUndo.Enabled = true;
    //Check if Redo is possible
    if (rtbText.CanRedo)
        mnuEditRedo.Enabled = true;
}
```

9. Insert the `Main()` method to launch form `GuidedPracticeExercise2_3.cs`. Set the form as the startup object for the project.

10. Run the project. Open or create an RTF file, select some text and select Edit, Copy to copy text to the Clipboard. Perform some cut, copy, and paste operations. Select Edit, Undo to undo the changes in the document (see Figure 2.39).



Exa veloping and
Imp ndows-based
App Microsoft Visual C#
.NE oft Visual Studio .NET

Creating User Services

Create a Windows Form by using the Windows Forms Designer.

- Add and set properties on a Windows Form.
- Create a Windows Form by using visual

FIGURE 2.39
The Edit menu provides standard editing commands such as Undo, Redo, Cut, Copy, and Paste.

If you have difficulty following this exercise, review the sections "The `MainMenu` Control" and "The `ContextMenu` Control" earlier in this chapter. Also, spend some time looking at the various methods and properties that are available for a `RichTextBox` control in the Properties window. Reading the text and examples in this chapter should help you relearn this material. After doing that review, try this exercise again.

The `StatusBar` Control

The `StatusBar` control is used to display information such as help messages and status messages. A `StatusBar` control is normally docked at the bottom of a form. When you add a `StatusBar` control to a form from the toolbox, make sure to set the z-order of the control by right-clicking the status bar and selecting Send to Back from the shortcut menu. When you do this, you do not see the status bar overlapping other controls at the bottom of the form.

NOTE

Layering Controls: Z-order *Z-order* specifies the visual layering of controls on a form along the form's z-axis, which specifies its depth. Controls are stacked in descending z-order value, with the control with the greatest z-order at the top of the stack and the control with the smallest z-order on the bottom of the stack. You can set the z-order of a control relative to its container control by right-clicking the control and selecting either Send to Back or Bring to Front from the shortcut menu.

One of the most important properties for a `StatusBar` control is the `Panels` property. `Panels` is a collection of `StatusBarPanel` objects. Panels divide a status bar area so that you can use each area to display a different type of information, such as the status of the date and time, page number, download progress, and the Caps Lock, Num Lock, and Insert keys. Table 2.31 summarizes the important members of the `StatusBarPanel` class.

TABLE 2.31

IMPORTANT MEMBERS OF THE `StatusBarPanel` CLASS

Member	*Type*	*Description*
`Alignment`	Property	Specifies the alignment of the text and icons within the panel. Can be one of the `Center`, `Left`, or `Right` values of the `HorizontalAlignment` enumeration.

continues

NOTE

The `AutoSize` Property The `StatusBarPanel` objects that have their `AutoSize` properties set to `StatusBarPanelAutoSize.Contents` have priority placement over the `StatusBarPanel` objects that have their `AutoSize` properties set to `StatusBarPanelAutoSize.Spring`. That is, a `StatusBarPanel` object with `AutoSize` set to `Spring` is shortened if a `StatusBarPanel` object with `AutoSize` set to `Contents` resizes itself to take more space on the status bar.

NOTE

Icon Positioning on a Status Bar Panel An icon is always positioned on the left side of the text in a panel, regardless of the text's alignment.

TABLE 2.31 *continued*

IMPORTANT MEMBERS OF THE `StatusBarPanel` CLASS

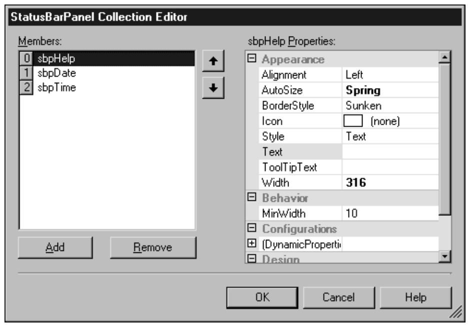
Member	*Type*	*Description*
`AutoSize`	Property	Specifies how the panel should size itself. This property can take the values `None`, `Contents`, and `Spring` from the `StatusBarPanelAutosize` enumeration.
`BorderStyle`	Property	Specifies the border style. `BorderStyle` can have the values `None`, `Raised`, and `Sunken` from the `StatusBarPanelBorderStyle` enumeration.
`Icon`	Property	Specifies the icon to be displayed in the status bar.
`Style`	Property	Specifies whether the `StatusBarPanel` object is set to `OwnerDraw` or `Text` (that is, system drawn). The `OwnerDraw` style can be use to give custom rendering to the `StatusBarPanel` object.
`ToolTipText`	Property	Specifies the ToolTip to show for the `StatusBarPanel` object.

STEP BY STEP

2.20 Creating a Status Bar for a Form

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_20`.
2. Follow steps 2 through 8 from Guided Practice Exercise 2.3.
3. Select the form `StepByStep2_20` by clicking its title bar. From the toolbox, double-click the `StatusBar` control to add it to the form. Name the `StatusBar` object `sbStatus` and clear its `Text` property. Change its `ShowPanels` property to `true`.
4. Select the status bar. Right-click it and select Send to Back from the shortcut menu.

5. Select the `Panels` property of the status bar. Click the ellipsis (…) button and create three `StatusBarPanel` objects, using the `StatusBarPanel` Collection Editor, as shown in Figure 2.40. Name the first object `sbpHelp`, change its `AutoSize` property to `Spring`, and empty its `Text` property. Name the second object `sbpDate`, change its `Alignment` property to `Right`, change `AutoSize` to `Contents`, change `ToolTipText` to `Current System Date`, and empty the `Text` property. Name the third object `sbpTime`, change its `Alignment` property to `Right`, change `AutoSize` to `Contents`, change `ToolTipText` to `Current System Time`, and empty the `Text` property.



StatusBarPanel Collection Editor

Members:

0 sbpHelp
1 sbpDate
2 sbpTime

sbpHelp Properties:

Appearance	
Alignment	Left
AutoSize	**Spring**
BorderStyle	Sunken
Icon	(none)
Style	Text
Text	
ToolTipText	
Width	**316**
Behavior	
MinWidth	10
Configurations	
(DynamicProperti	
Design	

FIGURE 2.40
The `StatusBarPanel` Collection Editor allows you to create and edit panels in a status bar.

6. Add a `Timer` control to the form, name it `tmrTimer`, change its `Enabled` property to `true`, and set the `Interval` property to `1000`. Double-click the `tmrTimer` object to add an event handler for its `Tick` event, and then add the following code to it:

```
private void tmrTimer_Tick(
    object sender, System.EventArgs e)
{
    DateTime dtNow = DateTime.Now;
    //display current date in the status bar panel
    sbpDate.Text = dtNow.ToLongDateString();
    //display current time in the status bar panel
    sbpTime.Text = dtNow.ToShortTimeString();
}
```

7. Insert the following event handling code in the program:

```
private void mnuItem_Select(
    object sender, System.EventArgs e)
{
    //Get the menu item sending the event
    MenuItem mnuItem = (MenuItem) sender;
    string strHelp;
    //Check which menu it is
    //and set the appropriate help text.
    switch(mnuItem.Text.Replace("&", ""))
    {
        case "New":
            strHelp = "Create a new document";
            break;
        case "Open...":
            strHelp = "Open an existing document";
            break;
        case "Save As...":
            strHelp = "Save the active document" +
              " with a new name";
```

continues

continued

```
            break;
        case "Exit":
            strHelp = "Quit the application";
            break;
        case "Undo":
            strHelp = "Undo the last action";
            break;
        case "Redo":
            strHelp = "Redo the last undone action";
            break;
        case "Cut":
       strHelp = "Cut the selection to the clipboard";
             break;
        case "Copy":
      strHelp = "Copy the selection to the clipboard";
            break;
        case "Paste":
            strHelp = "Insert clipboard contents";
            break;
        case "Color":
            strHelp = "Select a color";
            break;
        case "Font":
            strHelp = "Select a font";
            break;
        default:
            strHelp = "";
            break;
    }
    sbpHelp.Text = strHelp;
}
```

8. Select `mnuItem_Select` as the event handler for the `Select` events of all the menu items, including the top-level menus.

9. Add the following event handler in the code view:

```
//Reset the help text on status bar after the
//Menu is closed
protected override void OnMenuComplete(EventArgs e)
{
    sbpHelp.Text = "";
}
```

10. Insert the `Main()` method to launch form `StepByStep2_20`. Set the form as the startup object for the project.

11. Run the project. You should see the system date and time on the status bar. As you navigate through various menu items, you should see the description of menu items in the status bar (see Figure 2.41).

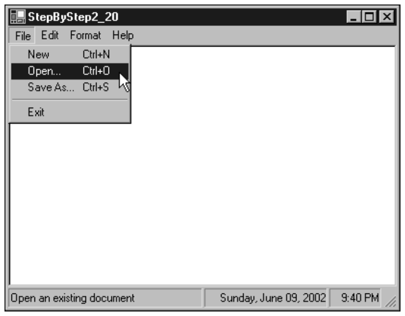


FIGURE 2.41
The `StatusBar` control is used to display various kinds of status information for an application.

The `ToolBar` Control

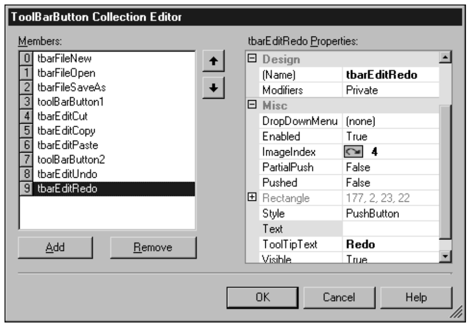
The `ToolBar` control can be used to create a Windows toolbar. It is normally docked on the top of the form, just below the menu bar. When you add a `ToolBar` control to form the toolbox, you should make sure to set the z-order of the control by right-clicking the toolbar and selecting Send to Back from the shortcut menu. When you do this, you do not see the toolbar overlapping other controls at the top of the form. Table 2.32 summarizes the important members of `ToolBar` class.

TABLE 2.32

IMPORTANT MEMBERS OF THE `ToolBar` CLASS

Member	*Type*	*Description*
`Buttons`	Property	Specifies a collection of `ToolBarButton` objects. Each `ToolbarButton` object represents a button on the `ToolBar` object.
`ButtonClick`	Event	Occurs when the toolbar button is clicked.
`ImageList`	Property	Specifies the `ImageList` object that stores the icons that will be displayed on the `ToolBarButton` objects.
`SendToBack()`	Method	Sends the toolbar to the back of the z-order.
`ShowToolTips`	Property	Specifies whether the toolbar should show ToolTips.

Normally the toolbar buttons represent shortcuts to tasks that could otherwise be done by selecting a menu option. When you respond to the `ButtonClick` event of a `ToolBar` object, you can simply invoke a corresponding menu item to accomplish a task, thereby reusing the efforts already invested in programming the menus. You can programmatically invoke a menu item by calling the `PerformClick()` method on a `MenuItem` object. So all you need to know is what menu to invoke for a particular toolbar button. The `ToolBarButton` control provides a useful property for this: the `Tag` property. You can use it to store any object that needs to be associated with a toolbar button. At program startup, you can use the `Tag` property of a `ToolBarButton` object to assign each button to a corresponding menu item. Step by Step 2.21 describes how to do this.



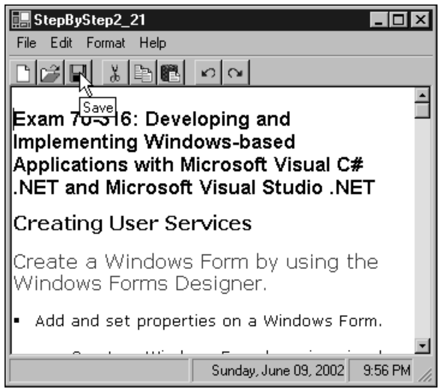
Members:

0	tbarFileNew
1	tbarFileOpen
2	tbarFileSaveAs
3	toolBarButton1
4	tbarEditCut
5	tbarEditCopy
6	tbarEditPaste
7	toolBarButton2
8	tbarEditUndo
9	tbarEditRedo

tbarEditRedo Properties:

Design	
(Name)	**tbarEditRedo**
Modifiers	Private
Misc	
DropDownMenu	(none)
Enabled	True
ImageIndex	**4**
PartialPush	False
Pushed	False
Rectangle	177, 2, 23, 22
Style	PushButton
Text	
ToolTipText	**Redo**
Visible	True

FIGURE 2.42
The ToolBarButton Collection Editor allows you to create and edit buttons in a toolbar.



Exam 70-316: Developing and Implementing Windows-based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET

Creating User Services

Create a Windows Form by using the Windows Forms Designer.

- Add and set properties on a Windows Form.

Sunday, June 09, 2002 | 9:56 PM

FIGURE 2.43
A toolbar contains buttons that carry out associated menu commands.

STEP BY STEP

2.21 Creating a Toolbar for a Form

1. Add a Windows form to existing project `316C02`. Name the form `StepByStep2_21`.

2. Follow steps 2 through 9 from Step by Step 2.20.

3. Select the form `StepByStep2_21` by clicking its title bar. From the toolbox, double-click the `ToolBar` control to add it to the form. Name the `ToolBar` object `tbarToolBar` and change both the height and width of its `ButtonSize` property to `16`. Change the `ShowToolTips` property to `true`.

4. Drop an `ImageList` (`imgToolBarIcons`) object on the form. To its `Images` property, add images for New, Open, Save, Cut, Copy, Paste, Undo, and Redo operations. Change the `ImageList` property of the `ToolBar` control to `imgToolBarIcons`.

5. Select the `ToolBar` control's `Buttons` property. Click the ellipsis (…) button to open the ToolBarButton Collection Editor Window. Using this window, add the buttons New, Open, Save, Cut, Copy, Paste, Undo, and Redo, and name them `tbarFileNew`, `tbarFileOpen`, `tbarFileSaveAs`, `tbarEditCut`, `tbarEditCopy`, `tbarEditPaste`, `tbarEditUndo`, and `tbarEditRedo`, respectively (see Figure 2.42). Select an image for each button from the `ImageIndex` property and give each button an appropriate `ToolTipText` setting. You can also add separators between the toolbar buttons by adding a `ToolBarButton` control and setting its `Style` property to `Separator`. The completed toolbar should look like the one shown in Figure 2.43.

6. Add the following code to the form's constructor:

```
public StepByStep2_21()
{
    //
    // Required for Windows Forms Designer support
    //
    InitializeComponent();
    // Store the references to menu items
    // in the toolbar buttons
    tbarFileNew.Tag = mnuFileNew;
    tbarFileOpen.Tag = mnuFileOpen;
    tbarFileSaveAs.Tag = mnuFileSaveAs;
    tbarEditCut.Tag = mnuEditCut;
    tbarEditCopy.Tag = mnuEditCopy;
    tbarEditPaste.Tag = mnuEditPaste;
    tbarEditUndo.Tag = mnuEditUndo;
    tbarEditRedo.Tag = mnuEditRedo;
}
```

7. Double-click the toolbar to add the following event handler for its `Click` event:

```
private void tbarToolBar_ButtonClick(object sender,
   System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
    ToolBarButton tbarButton = e.Button;
    //Get related menu item from the
    //toolbar buton's tag property
    MenuItem mnuItem = (MenuItem) tbarButton.Tag;

    //Generate the click event for related menu item
    mnuItem.PerformClick();
}
```

8. Insert the `Main()` method to launch form `StepByStep2_21`. Set this form as the startup object for the project.

9. Run the project, and you should see the toolbar. Click the various toolbar buttons to do desired tasks (refer to Figure 2.43).

Menu toolbar buttons map directly to menu items. Clicking a menu toolbar button is the same as clicking on its corresponding menu item (or pressing the hotkey for that item).

Disabling a menu item disables its toolbar button as well, but it does not change the appearance of the button. If you want the button to “look” disabled, you must do it programmatically.

CREATING MDI APPLICATIONS

So far in this chapter, you have created only single-document interface (SDI) applications. These applications allow a user to work with only one window at a time. Several large Windows applications, such as Microsoft Excel and Visual Studio .NET, allow users to work with several open windows at the same time. These applications are called *multiple-document interface* (MDI) applications. The main application window of an MDI application acts as the parent window, which can open several child windows. You need to know the following main points about an MDI application:

- Child windows are restricted to their parent window (that is, you cannot move a child window outside its parent window).
- The parent window can open several types of child windows. As an example of an MDI application, Visual Studio .NET allows you to work with several types of document windows at the same time.
- The child windows can be opened, closed, maximized, or minimized independently of each other, but when the parent window is closed, the child windows are automatically closed.
- The MDI frame should always have a menu; one of the menus that a user always expects to see in an MDI application is the Window menu (see Figure 2.44), which allows the user to manipulate various windows that are open in an MDI container form.

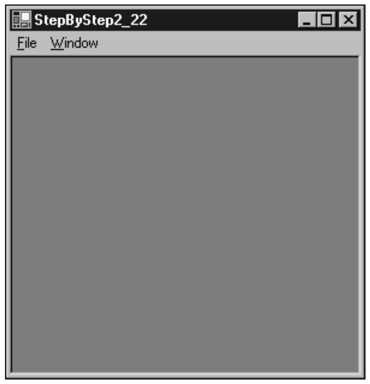


FIGURE 2.44
An MDI application typically has a Window menu item.

The Windows Forms in an MDI application are also created by using the standard `System.Windows.Forms.Form` class. To create an MDI parent form, you create a regular Windows form and change its `IsMdiContainer` property to `true`. To create an MDI child form, you create a regular Windows form and assign the name of the parent MDI object to the `MdiParent` property. Table 2.33 summarizes the important members of the `Form` class that are related to the MDI applications.

TABLE 2.33

IMPORTANT MEMBERS OF THE `Form` CLASS THAT ARE RELATED TO MDI APPLICATIONS

Member	*Type*	*Description*
`ActiveMdiChild`	Property	Identifies the currently active MDI child window
`IsMdiContainer`	Property	Indicates whether the form is a container for MDI child forms
`MdiChildActivate`	Event	Occurs when anMDI child form is activated or closed within an MDI application
`MdiChildren`	Property	Specifies an array of forms that represent the MDI child form of the form
`MdiParent`	Property	Specifies the MDI parent form for the current form
`LayoutMdi()`	Method	Arranges the MDI child forms within an MDI parent form

Step by Step 2.22 shows how to create an MDI application. In it you can create a form that is similar to the one created in Step by Step 2.18 and you can use it as an MDI child window.

You learn the following from Step by Step 2.22:

- How to create an MDI parent form
- How to create MDI child forms
- How to convert an existing SDI application to an MDI application
- How to merge menus between MDI parent and child windows
- How to create Windows menus that allow users to load and rearrange MDI child forms

STEP BY STEP

2.22 Creating an MDI Application

1. Add a Windows form to existing project `316C02`. Name the form `MdiChild`.
2. Follow steps 2 to 27 from Step by Step 2.18.
3. Select the main menu of the form and change the `MergeType` property of the File menu to `MergeItems`. Select the File, New and File, Open menus and change their `MergeType` properties to `Remove`.
4. Rename the File, Exit menu item `&Close`. Change the `MergeOrder` properties for the File, Save As; File, Close; Format; and Help menu items to `5`, `3`, `10`, and `30`, respectively. Change the `MergeOrder` property for the separator in the File menu to `4`.
5. Switch to the code view. After the default constructor code, add another constructor with the following code:

```
public MdiChild(string fileName)
{
    //
    // Required for Windows Forms Designer support
    //
    InitializeComponent();
    rtbText.LoadFile(fileName,
        RichTextBoxStreamType.RichText);
}
```

6. Add another Windows form to existing project `316C02`. Name the form `StepByStep2_22` and change its `IsMdiContainer` property to `true`.
7. Add a `MainMenu` component to the form. Create a top-level menu item, change its `Text` property to `&File`, and name it `mnuFile`. Add the following menu items to it: `&New` (`mnuFileNew`), &`Open...` (`mnuFileOpen`), separator, and `E&xit` (`mnuFileExit`). Change the menu items' `MergeOrder` properties to `1`, `2`, `6`, and `7`, respectively.

8. Add another top-level menu. Change the `Text` property to `&Window`, and the `Name` property to `mnuWindow`, and the `MdiList` property to `true`. Add the following menu items to it: `Tile &Horizontally` (`mnuWindowTileHorizintally`), `Tile &Vertically` (`mnuWindowTileVertically`), and `&Cascade` (`mnuWindowCascade`).

9. Double-click the File, New menu item and add the following event handler to its `Click` event:

```
private void mnuFileNew_Click(
    object sender, System.EventArgs e)
{
    //create a new instance of child window
    MdiChild frmMdiChild = new MdiChild();
    //set its MdiParent
    frmMdiChild.MdiParent = this;
    frmMdiChild.Text = "New Document";
    //Display the child window
    frmMdiChild.Show();
}
```

10. Double-click the File, Open menu item and add the following event handler to it:

```
private void mnuFileOpen_Click(
    object sender, System.EventArgs e)
{
    //Allow to select only *.rtf files
    dlgOpenFile.Filter =
        "Rich Text Files (*.rtf)|*.rtf";
    if(dlgOpenFile.ShowDialog() == DialogResult.OK)
    {
        //create the child form by
        //loading the given file in it
        MdiChild frmMdiChild =
           new MdiChild(dlgOpenFile.FileName);
        //Set the current for as its parent
        frmMdiChild.MdiParent = this;
        //Set the file's title bar text
        frmMdiChild.Text = dlgOpenFile.FileName;
        //display the form
        frmMdiChild.Show();
    }
}
```

continues

11. Add the following event handlers to `Click` events of other menu items, as shown in steps 9 to 10:

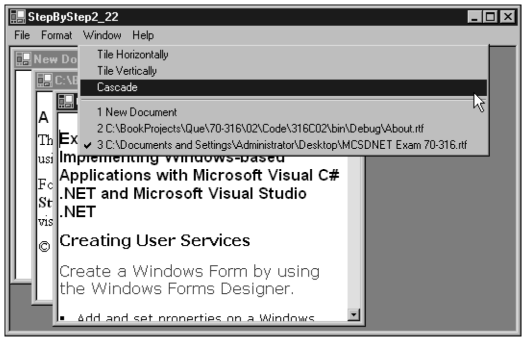
```
private void mnuFileExit_Click(
     object sender, System.EventArgs e)
{
    //Close the parent window
    this.Close();
}
private void mnuWindowTileHorizontally_Click(
     object sender, System.EventArgs e)
{
    //Tile child windows horizontally
    this.LayoutMdi(MdiLayout.TileHorizontal);
}
private void mnuWindowTileVertically_Click(
    object sender, System.EventArgs e)
{
    //Tile child windows vertically
    this.LayoutMdi(MdiLayout.TileVertical);
}
private void mnuWindowCascade_Click(
    object sender, System.EventArgs e)
{
    //cascade
    this.LayoutMdi(MdiLayout.Cascade);
}
```

12. Add thefollowing event handler to the `Popup` event of the `mnuWindow` menu item:

```
private void mnuWindow_Popup(
    object sender, System.EventArgs e)
{
    //code to enable and disable Window menu items
    //depending on if any child windows are open
    if (this.MdiChildren.Length > 0)
    {
        mnuWindowTileHorizontally.Enabled = true;
        mnuWindowTileVertically.Enabled = true;
        mnuWindowCascade.Enabled = true;
    }
    else
    {
        mnuWindowTileHorizontally.Enabled = false;
        mnuWindowTileVertically.Enabled = false;
        mnuWindowCascade.Enabled = false;
    }
}
```

13. Insert the `Main()` method to launch form `StepByStep2_22`. Set the form as the startup object for the project.

14. Run the project. From the File menu, open a new document, and then also open an existing document by selecting File, Open. Click the Window menu and select various options to arrange the child windows (see Figure 2.45).


StepByStep2_22
File
Format
Window
Help
Tile Horizontally
Tile Vertically
Cascade
1 New Document
2 C:\BookProjects\Que\70-316\02\Code\316C02\bin\Debug\About.rtf
✓ 3 C:\Documents and Settings\Administrator\Desktop\MCSDNET Exam 70-316.rtf
New Do
C:\B

Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET

Creating User Services

Create a Windows Form by using the Windows Forms Designer.

FIGURE 2.45
When you are working with an MDI application, you can use the commands from the Window menu to switch between windows or documents.

The default `MergeType` property for the `MenuItem` objects is `Add`. This means that the `MenuItem` objects in Step by Step 2.22 are added together on an MDI parent window. If you don't want to include some of the menu items, you can set their `MergeType` properties to `Remove`. The `MergeOrder` properties for the `MenuItem` objects specify the order in which they appear on the parent MDI form.

There is another interesting property of the `MenuItem` object that is used with MDI applications: the `MdiList` property. When this property is set to `true`, the `MenuItem` object is populated with a list of MDI child windows that are displayed within the associated form.

REVIEW BREAK

- There are two primary types of menus. The main menu is used to group all the available commands and option in a Windows application, and a context menu is used to specify a relatively smaller list of options that apply to a control, depending on the application's current context.

continues

continued

- You can make keyboard navigation among menu items possible by associating hotkeys in the `Text` property of the menu items. You can also associate shortcut keys, which directly invoke commands, with a menu.
- The `Clipboard` object consists of two public static methods, `GetDataObject()` and `SetDataObject()`, which get and set the data from the Clipboard.
- The `StatusBar` control creates a standard Windows status bar in an application. You can use `StatusBar` to display various messages and help text to the user.
- You can use toolbars to create a set of small buttons that are identified by icons. Toolbars generally provide shortcuts to operations that are available in the application's main menu. Toolbars are common in Windows applications and make an application simple to use.
- An MDI application allows multiple documents or windows to be open at the same time.

CHAPTER SUMMARY

Using Visual Studio .NET, you can add controls to a form in two ways: Either you can use the convenient Windows Forms Designer or you can hand code the controls and load them dynamically in application.

Visual Studio .NET comes with a whole array of Windows Forms controls, and you can set their properties at design time and run-time. Visual Studio .NET makes event handling easy by automatically creating event handlers for controls.

Visual Studio .NET comes with full-fledged menu controls that are needed by almost every real-time application. Visual Studio .NET allows you to create main menus as well as context menus. Visual Studio .NET's MDI forms support is extensive, allowing you to merge menu items with menus of child windows.

This chapter presents the rich library of Windows forms controls. It discusses most of the common Windows controls used by the applications. In fact, the .NET Framework Software Development Kit (SDK) also allows you to create your own controls. In addition, a large number of controls are available from several third-party control vendors. Chapter 4, "Creating and Managing .NET Components and Assemblies" explains how to create user controls and add them to the list of available controls.

KEY TERMS

- Clipboard
- context menu
- main menu
- MDI application
- SDI application
- tab order
- ToolTip
- z-order

Exercises

2.1 Adding ToolTips to Controls

In this exercise you will learn how to set ToolTips for a form. A few Windows forms controls, such as `TabPage`, `ToolBarButton`, and `StatusBarPanel`, have `ToolTipText` properties for showing ToolTips. But other commonly used controls do not have any built-in properties to which ToolTips text can be assigned. This exercise introduces the `ToolTip` component. When you drag the `ToolTip` component from the toolbox into the Windows form, this component provides a new property for all the controls on the form. This new property is named `ToolTip on` *`tooltipComponentName`* and can be edited through the Properties window. When the form executes, for each of its controls it shows the value of the `ToolTip on` *`Tooltip ComponentName`* property as the ToolTip for the control.

Estimated time: 25 minutes

1. Create a new Visual C# .NET Windows application in the Visual Studio .NET IDE.
2. Add a new form to your Visual C# .NET project. Name it `Exercise2_1.cs`. Change the `Text` property of the form to `Find`, change `FormBorderStyle` to `Fixed3D`, and change `TopMost` to `true`. Set the `MaximizeBox`, `MimimizeBox`, and `ShowInTaskBar` properties to `false`.
3. Place a ToolTip component on the form and name it `tTip`. It is added to the component tray.
4. Place one `Label` control (keep its default name), one `TextBox` control (`txtTextToFind`), two `Button` controls (`btnFind` and `btnCancel`), one `CheckBox` control (`chkMatchCase`), and a group box with two `RadioButton` controls (`rbUp` and `rbDown`) on the form. Arrange the controls as shown in Figure 2.46.

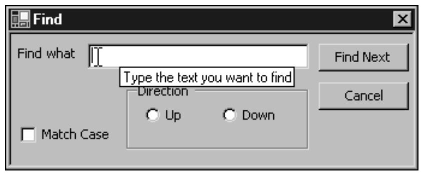


FIGURE 2.46
The Windows `ToolTip` component displays text when the user points at controls.

5. A property named `ToolTip on tTip` is added to all the controls placed on the form. Modify this property to enter an appropriate ToolTip message for each control on the form.
6. Insert the `Main()` method and set the form as the startup object of the project.
7. Run the project. The ToolTip message is displayed when you rest the mouse pointer over the control (refer to Figure 2.46).

2.2 Dynamically Creating Menu Items

While creating menus in Windows applications, you are often required to add menu items dynamically. This exercise shows you how to create menu items dynamically, and it shows a list of the recent files opened by the application.

Estimated time: 40 minutes

1. Add a new form to your Visual C# .NET project. Name it `Exercise2_2.cs`.
2. Place a `MainMenu` control (`mnuMainMenu`) and an `OpenFileDialog` control (`dlgOpenFile`) on the form.
3. Using the menu designer, add a top-level menu item, set its `Text` property to `&File`, and name it `mnuFile`. Add three menu items: `&Open` (`mnuFileOpen`), `Recent &Files` (`mnuFileRecentFiles`), and `E&xit` (`mnuExit`).

APPLY YOUR KNOWLEDGE

4. Change the `Menu` property of the form to `mnuMainMenu`.

5. Add the following code in the code view, just before the constructor:

```
//Store recently used file
//list in ArrayList
private ArrayList alRecentFiles;
private System.Windows.Forms.MainMenu
   mnuMainMenu;
//Maximum number to files to store
private const int intListSize = 4;
```

6. Add the following code in the constructor:

```
//Create ArrayList of given size
alRecentFiles = new ArrayList(intListSize);
```

7. Attach the default `Click` event handlers for the `mnuFileExit`, `mnuFileOpen`, and `mnuFileRecentFiles` menu items and add the following code to handle the `Click` events of the menu items:

```
private void mnuFileExit_Click(
   object sender, System.EventArgs e)
{
    this.Close();
}

private void mnuFileOpen_Click(
   object sender, System.EventArgs e)
{
    if(dlgOpenFile.ShowDialog() ==
        DialogResult.OK)
    {
        //Find if the file already
        //exists in the ArrayList
        int pos = alRecentFiles.IndexOf(
             dlgOpenFile.FileName);
        //If it exists then remove it
        if (pos >= 0)
            alRecentFiles.RemoveAt(pos);

        //If you have exceeded the size
        //of ArrayList
        //delete the oldest item from it
        if (alRecentFiles.Count >=
            intListSize)
            alRecentFiles.RemoveAt(
               intListSize-1);
            //Add the recently opened file to
            //the queue of recent files
            alRecentFiles.Insert(0,
               dlgOpenFile.FileName);

            //do some processing here...
            MessageBox.Show(
               "You selected to open: " +
               dlgOpenFile.FileName,
               "File Opened",
MessageBoxButtons.OK,
               MessageBoxIcon.Information);
    }
}
private void mnuFileRecentFilesItem_Click(
    object sender, System.EventArgs e)
{
    MenuItem mnuItem = (MenuItem) sender;
    //do some processing here
    MessageBox.Show("You selected to open: "
+
       mnuItem.Text.Substring(2),
       "File Opened", MessageBoxButtons.OK,
       MessageBoxIcon.Information);
}
```

8. Add the `Popup` event handler for the `mnuFile` menu item:

```
private void mnuFile_Popup(
    object sender, System.EventArgs e)
{
    //Check if there are any file
    //names in the list
    if (alRecentFiles.Count > 0)
    {
      //Clear old recent file list
      mnuFileRecentFiles.MenuItems.Clear();
        //Use this number to add keyboard
        //shortcut to menu items
        //Most recent file has short of 1,
        //next file has shortcut of 2..
        int intFileCount = 1;
        foreach (string fileName in
             alRecentFiles)
        {
            //Create a menu item to create
            //in File - Recent Files menu
            MenuItem mnuItem =
                new MenuItem();
```

APPLY YOUR KNOWLEDGE

```
                //Set the MenuItem text with
                //a shortcut key
                mnuItem.Text = String.Format(
                  "&{0} {1}", intFileCount++,
                  fileName);
                //attach an event handler
                //to this menu item
                mnuItem.Click += new
                 System.EventHandler(
                 mnuFileRecentFilesItem_Click);
                //Add the recently used file in
                // the File - Recent files menu
              mnuFileRecentFiles.MenuItems.Add(
                   mnuItem);
            }
            //Now that I have some files in
            //the File - Recent Files menu,
            //Enable it
            mnuFileRecentFiles.Enabled = true;
        }
        else
            //If there are no recent files,
            //disable the menu item
            mnuFileRecentFiles.Enabled = false;
    }
```

9. Insert the `Main()` method and set the form as the startup object of the project.

10. Run the project. Open a few files by selecting File, Open, and then select the Recent Files menu item. The recently opened files are added as submenu items to the Recent Files menu item. Also, the files appear in order in terms of how recently they were opened (see Figure 2.47).

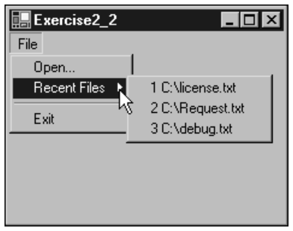


FIGURE 2.47
You can display a list of recently used files by dynamically creating menu items.

Review Questions

1. Where in a form can controls be placed? What are the two ways to add controls?
2. What is the shortcut way of creating an event handler for the default event of a control?
3. When should you choose a `ComboBox` control instead of a `ListBox` control in an application?
4. What different modes of selection are possible in a list box?
5. What are the roles of the `TabIndex` and `TabStop` properties?
6. How can you create modal and modeless dialog boxes?
7. What are the different styles for drawing combo boxes?
8. What is the function of the `Tag` property in a control?
9. When does the `Popup` event of a menu occur? What is the main reason this event is used?
10. What is the difference between the `DateTimePicker` and `MonthCalendar` controls? What do you need to do in order to display a custom format in a `DateTimePicker` control?
11. What is an MDI application?
12. How do you merge menu items of a child window with an MDI window?

OBJECTIVES

This chapter covers the following Microsoft-specified objectives for the "Creating User Services" section of Exam 70-316, "Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET":

Implement error handling in the UI.

- **Create and implement custom error messages**
- **Create and implement custom error handlers.**
- **Raise and handle errors.**

▶ When you run a Windows application, it may encounter problems that you thought would not occur. For example, the database server is down, a file is missing, or a user has entered improper values. A good Windows application must recover gracefully from this problem rather than abruptly shut down. This exam objective covers the use of exception handling to create robust and fault-tolerant applications. The Microsoft .NET Framework provides some predefined exception classes to help you catch these exceptional situations in your programs. It allows you to create your own exception handling classes and error messages that are specific to your application.

Validate user input.

▶ Garbage in results in garbage out. The best place to avoid incorrect data in an application is at the source—right where the data enters. The Windows Forms library provides an `ErrorProvider` component that can be used to display helpful error messages and error icons if data that is entered is incorrect. This exam objective covers the `ErrorProvider` component and various other input-validation techniques.

CHAPTER 3

Error Handling for the User Interface

OUTLINE

STUDY STRATEGIES

- Review the "Exception Handling Statements" and the "Best Practices for Exception Handling" sections of the Visual Studio .NET Combined Help Collection. The Visual Studio .NET Combined Help Collection is installed as part of the Visual Studio .NET installation.
- Experiment with code that uses the `try`, `catch`, and `finally` blocks. Use these blocks with various combinations and inspect the differences in your code's output.
- Know how to create custom exception classes and custom error messages; learn to implement them in a program.
- Experiment with the `ErrorProvider` component, the `Validating` event, and other validation techniques. Use these tools in various combinations to validate data that is entered in controls.

INTRODUCTION

The .NET Framework uses the Windows structured exception handling model. Exception handling is an integral part of the .NET Framework that allows the Common Language Runtime (CLR) and your code to throw exceptions across languages and machines. Visual C# .NET helps you fire and handle these exceptions with the help of `try`, `catch`, `finally`, and `throw` statements. The Framework Class Library (FCL) provides a huge set of exception classes for dealing with various unforeseen conditions in the normal execution environment. If you feel the need to create custom exception classes to meet the specific requirements of an application, you can do so by deriving from the `ApplicationException` class.

In every program data must be validated before the program can proceed with further processing and storage of the input data. In this chapter I discuss the various techniques you can use to validate data and maintain the integrity of an application. This isn't just a matter of making sure that your application delivers the proper results; if you don't validate input, your application might represent a serious security hole in your systems.

UNDERSTANDING EXCEPTIONS

An *exception* occurs when a program encounters any unexpected problems such as running out of memory or attempting to read from a file that no longer exists. These problems are not necessarily caused by programming errors but mainly occur due to violations of certain assumptions that are made about the execution environment.

When a program encounters an exception, its default behavior is to throw the exception, which generally translates to abruptly terminating the program after displaying an error message. This is not a characteristic of a robust application and does not make your program popular with users. Your program should be able to handle these exceptional situations and, if possible, gracefully recover from them. This is called *exception handling*. Proper use of exception handling can make programs robust and easy to develop and maintain. However, if you do not use exception handling properly, you might end up having a program that performs poorly, is harder to maintain, and may potentially mislead its users.

Step by Step 3.1 demonstrates how an exception may occur in a program. Later in this chapter I explain how to handle these exceptions.

STEP BY STEP

3.1 Exceptions in Windows Applications

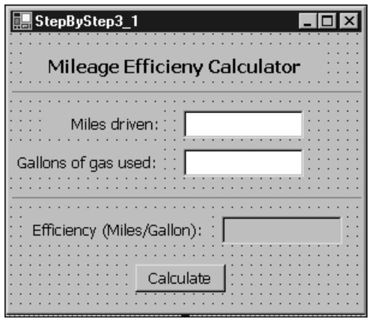
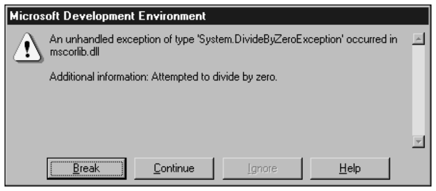


FIGURE 3.1
The mileage efficiency calculator does not implement any error handling for the user interface.

1. Create a new C# Windows application project in the Visual Studio .NET Integrated Development Environment (IDE). Name the project `316C03`.
2. Add a new Windows form to the project. Name it `StepByStep3_1`.
3. Place three `TextBox` controls (`txtMiles`, `txtGallons`, and `txtEfficiency`) and a `Button` control (`btnCalculate`) on the form and arrange them as shown in Figure 3.1. Add `Label` controls as necessary.
4. Add the following code to the `Click` event handler of `btnCalculate`:

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
    //this code has no error checking. If something
    //goes wrong at run time,
    //it will throw an exception
    decimal decMiles =
        Convert.ToDecimal(txtMiles.Text);
    decimal decGallons =
        Convert.ToDecimal(txtGallons.Text);
    decimal decEfficiency = decMiles/decGallons;
    txtEfficiency.Text =
        String.Format("{0:n}", decEfficiency);
}
```

5. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.
6. Run the project. Enter values for miles and gallons and click the Calculate button. The program calculates the mileage efficiency, as expected. Now enter the value `0` in the Gallons of Gas Used field and run the program. The program abruptly terminates after displaying an error message (see Figure 3.2).

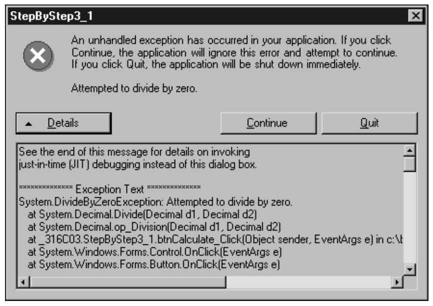

Microsoft Development Environment

An unhandled exception of type 'System.DivideByZeroException' occurred in mscorlib.dll

Additional information: Attempted to divide by zero.

FIGURE 3.2
The development environment gives you a chance to analyze the problem when an exception occurs.

When you run the program created in Step by Step 3.1 from the IDE and the program throws an exception, the IDE gives you options to analyze the problem by debugging the program. In Step by Step 3.1, if you had instead run the program by launching the project's `.exe` file from Windows Explorer, the program would have terminated after displaying a message box with an error message and some debugging information (see Figure 3.3).



```
An unhandled exception has occurred in your application. If you click
Continue, the application will ignore this error and attempt to continue.
If you click Quit, the application will be shut down immediately.

Attempted to divide by zero.
```

```
See the end of this message for details on invoking
just-in-time (JIT) debugging instead of this dialog box.

************** Exception Text **************
System.DivideByZeroException: Attempted to divide by zero.
   at System.Decimal.Divide(Decimal d1, Decimal d2)
   at System.Decimal.op_Division(Decimal d1, Decimal d2)
   at _316C03.StepByStep3_1.btnCalculate_Click(Object sender, EventArgs e) in c:\
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
```

FIGURE 3.3
When a program is executed outside the Visual Studio .NET environment, debugging information is displayed when an exception is thrown.

From the CLR's point of view, an exception is an object that encapsulates information about the problems that occur during program execution. The FCL provides two categories of exceptions:

- **`ApplicationException`**—Represents exceptions thrown by the applications
- **`SystemException`**—Represents exceptions thrown by the CLR

Both of these exception classes derive from the base class `Exception`, which implements the common functionality for exception handling. Neither the `ApplicationException` class nor the `SystemException` class adds any new functionality to the `Exception` class; they exist just to differentiate exceptions in applications from exceptions in the CLR. The classes derived from `Exception` share some important properties, as listed in Table 3.1

TABLE 3.1

IMPORTANT MEMBERS OF THE `Exception` CLASS

Member	*Type*	*Description*
`HelpLink`	Property	Specifies the uniform resource locator (URL) of the help file associated with this exception.
`InnerException`	Property	Specifies an exception associated with this exception. This property is helpful when a series of exceptions are involved. Each new exception can preserve the information about the previous exception by storing it in the `InnerException` property.
`Message`	Property	Specifies textual information that indicates the reason for the error and provides possible resolutions.
`Source`	Property	Specifies the name of the application that causes the error.

continues

EXAM TIP

Floating-Point Types and Exceptions Operations that involve floating-point types never produce exceptions. Instead, in exceptional situations, floating-point operations are evaluated by using the following rules:

- If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN (not a number).

TABLE 3.1 *continued*

IMPORTANT MEMBERS OF THE `Exception` CLASS

Member	*Type*	*Description*
`StackTrace`	Property	Specifies where an error has occurred. If the debugging information is available, the stack trace includes the name of the source file and the program line number.
`TargetSite`	Property	Represents the method that throws the current exception.

HANDLING EXCEPTIONS

Implement error handling in the user interface:

- **Create and implement custom error messages.**
- **Raise and handle errors.**

Abruptly terminating a program when an exception occurs is not a good idea. An application should be able to handle an exception and, if possible, try to recover from it. If recovery is not possible, you can have the program take other steps, such as notify the user and then gracefully terminate the application.

The .NET Framework allows exception handling to interoperate among languages and across machines. You can catch exceptions thrown by code written in one .NET language in a different .NET language. The .NET framework also allows you to handle exceptions thrown by legacy Component Object Model (COM) applications and legacy non-COM Windows applications.

Exception handling is such an integral part of the .NET framework that when you look for a method reference in the product documentation, there is always a section that specifies what exceptions a call to that method might throw.

You can handle exceptions in Visual C# .NET programs by using a combination of exception handling statements: `try`, `catch`, `finally`, and `throw`.

The `try` Block

You should place the code that might cause exceptions in a `try` block. A typical `try` block looks like this:

```
try
{
    //code that may cause exception
}
```

You can place any valid C# statements inside a `try` block, including another `try` block or a call to a method that places some of its statements inside a `try` block. The point is, at runtime you may have a hierarchy of `try` blocks placed inside each other. When an exception occurs at any point, rather than executing any further lines of code, the CLR searches for the nearest `try` block that encloses this code. The control is then passed to a matching `catch` block (if any) and then to the `finally` block associated with this `try` block.

A `try` block cannot exist on its own; it must be immediately followed by either one or more `catch` blocks or a `finally` block.

The `catch` Block

You can have several `catch` blocks immediately following a `try` block. Each `catch` block handles an exception of a particular type. When an exception occurs in a statement placed inside the `try` block, the CLR looks for a matching `catch` block that is capable of handling that type of exception. A typical `try-catch` block looks like this:

```
try
{
    //code that may cause exception
}
catch(ExceptionTypeA)
{
    //Statements to handle errors occurring
    //in the associated try block
}
catch(ExceptionTypeB)
{
    //Statements to handle errors occurring
    //in the associated try block
}
```

NOTE

Exception Handling Hierarchy If there is no matching catch block, an unhandled exception results. The unhandled exception is propagated back to its caller code. If the exception is not handled there, it propagates further up the hierarchy of method calls. If the exception is not handled anywhere, it goes to the CLR, whose default behavior is to terminate the program immediately.

The formula the CLR uses to match the exception is simple: While matching it looks for the first catch block with either the exact same exception or any of the exception's base classes. For example, a DivideByZeroException exception would match any of these exceptions: DivideByZeroException, ArithmeticException, SystemException, and Exception. In the case of multiple catch blocks, only the first matching catch block is executed. All other catch blocks are ignored.

When you write multiple catch blocks, you need to arrange them from specific exception types to more general exception types. For example, the catch block for catching a DivideByZeroException exception should always precede the catch block for catching a ArithmeticException exception. This is because the DivideByZeroException exception derives from ArithmeticException and is therefore more specific than the latter. The compiler flags an error if you do not follow this rule.

A try block need not necessarily have a catch block associated with it, but if it does not, it must have a finally block associated with it.

STEP BY STEP

3.2 Handling Exceptions

1. Add a new Windows form to the project. Name it StepByStep3_2.
2. Create a form similar to the one created in Step by Step 3.1 (refer to Figure 3.1), with the same names for the controls.
3. Add the following code to the Click event handler of btnCalculate:

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
    //put all the code that may require graceful
    //error recovery in a try block
    try
    {
        decimal decMiles =
            Convert.ToDecimal(txtMiles.Text);
        decimal decGallons =
            Convert.ToDecimal(txtGallons.Text);
        decimal decEfficiency = decMiles/decGallons;
        txtEfficiency.Text =
            String.Format("{0:n}", decEfficiency);
    }
    // try block should at least have one catch or a
    // finally block. catch block should be in order
    // of specific to the generalized exceptions
    // otherwise compilation generates an error
    catch (FormatException fe)
    {
        string msg = String.Format(
             "Message: {0}\n Stack Trace:\n {1}",
             fe.Message, fe.StackTrace);
        MessageBox.Show(msg, fe.GetType().ToString());
    }
    catch (DivideByZeroException dbze)
    {
        string msg = String.Format(
             "Message: {0}\n Stack Trace:\n {1}",
             dbze.Message, dbze.StackTrace);
        MessageBox.Show(
            msg, dbze.GetType().ToString());
    }
    //catches all CLS-compliant exceptions
    catch(Exception ex)
    {
        string msg = String.Format(
            "Message: {0}\n Stack Trace:\n {1}",
            ex.Message, ex.StackTrace);
        MessageBox.Show(msg, ex.GetType().ToString());
    }
    //catches all other exceptions including
    //the NON-CLS compliant exceptions
    catch
    {
        //just rethrow the exception to the caller
        throw;
    }
}
```

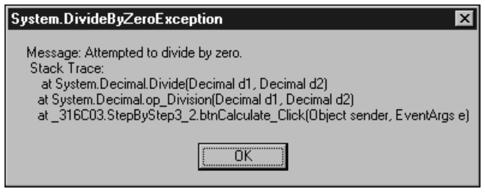
4. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.

5. Run the project. Enter values for miles and gallons and click the Calculate button. The program calculates the mileage efficiency, as expected. Now enter the value `0` in the Gallons of gas used field and run the program. Instead of abruptly terminating as in the earlier case, the program shows a message about the `DivideByZeroException` exception, as shown in Figure 3.4, and it continues running. Now enter some alphabetic characters instead of number in the fields and click the Calculate button.

continues

EXAM TIP

CLS- and Non-CLS-Compliant Exceptions All languages that follow the Common Language Specification (CLS) throw exceptions of type `System.Exception` or a type that derives from `System.Exception`. A non-CLS-compliant language may throw exceptions of other types, too. You can catch those types of exceptions by placing a general `catch` block (that does not specify any exception) with a `try` block. In fact, a general `catch` block can catch exceptions of all types, so it is the most generic of all `catch` blocks and should be the last `catch` block among the multiple `catch` blocks associated with a `try` block.



```
Message: Attempted to divide by zero.
 Stack Trace:
   at System.Decimal.Divide(Decimal d1, Decimal d2)
  at System.Decimal.op_Division(Decimal d1, Decimal d2)
  at _316C03.StepByStep3_2.btnCalculate_Click(Object sender, EventArgs e)
```

FIGURE 3.4
To get information about an exception, you can catch the `Exception` object and access its `Message` property.

NOTE

`checked` and `unchecked` Visual C# .NET provides the `checked` and `unchecked` keywords, which can be used to enclose a block of statements (for example, `checked {a = c/d}`) or as an operator when you supply parameters enclosed in parentheses (for example, `unchecked(c/d)`). The `checked` keyword enforces checking of any arithmetic operation for overflow exceptions. If constant values are involved, they are checked for overflow at compile time. The `unchecked` keyword suppresses the overflow checking and instead of raising an `OverflowException` exception, the `unchecked` keyword returns a truncated value in case of overflow.

If `checked` and `unchecked` are not used, the default behavior in C# is to raise an exception in case of overflow for a constant expression or truncate the results in case of overflow for the nonconstant expressions.

NOTE

Do Not Use Exceptions to Control the Normal Flow of Execution Using exceptions to control the normal flow of execution can make your code difficult to read and maintain because the use of `try-catch` blocks to deal with exceptions forces you to fork the regular program logic between two separate locations—the `try` block and the `catch` block.

continued

This time you get a `FormatException` exception, and the program continues to run. Now try entering very large values in both the fields. If the values are large enough, the program encounters an `OverflowException` exception, but because the program is catching all types of exceptions, it continues running.

The program in Step by Step 3.2 displays a message box when an exception occurs; the `StackTrace` property lists the methods in the reverse order of their calling sequence. This helps you understand the logical flow of the program. You can also place any appropriate error handling code in place, and you can display a message box.

When you write a `catch` block that catches exceptions of type `Exception`, the program catches all CLS-compliant exceptions. This includes all exceptions, unless you are interacting with legacy COM or Windows 32-bit Application Programming Interface (Win32 API) code. If you want to catch all kinds of exceptions, whether CLS-compliant or not, you can place a `catch` block with no specific type. A `catch` block like this must be the last `catch` block in the list of `catch` blocks because it is the most generic one.

You might be thinking that it is a good idea to catch all sorts of exceptions in your code and suppress them as soon as possible. But it is not a good idea. A good programmer catches an exception in code only if he or she can answer yes to one or more of the following questions:

- Will I attempt to recover from this error in the `catch` block?
- Will I log the exception information in the system event log or another log file?
- Will I add relevant information to the exception and rethrow it?
- Will I execute cleanup code that must run even if an exception occurs?

If you answer no to all these questions, then you should not catch the exception but rather just let it go. In that case, the exception propagates up to the calling code, and the calling code might have a better idea of how to handle the exception.

The `throw` Statement

A `throw` statement explicitly generates an exception in code. You use `throw` when a particular path in code results in an anomalous situation.

You should not throw exceptions for anticipated cases such as the user entering an invalid username or password; instead, you can handle this in a method that returns a value indicating whether the login is successful. If you do not have the correct permissions to read records from the user table and you try to read those records, an exception is likely to occur because a method for validating users should normally have read access to the user table.

There are two ways you can use the `throw` statement. In its simplest form, you can just rethrow the exception in a `catch` block:

```
catch(Exception e)
{
    //TODO: Add code to create an entry in event log
    throw;
}
```

This usage of the `throw` statement rethrows the exception that was just caught. It can be useful in situations in which you don't want to handle the exception yourself but would like to take other actions (for example, recording the error in an event log, sending an email notification about the error) when an exception occurs and then pass the exception as-is to its caller.

The second way to use the `throw` statement is to use it to throw explicitly created exceptions, as in this example:

```
string strMessage =
    "EndDate should be greater than the StartDate";
ArgumentOutOfRangeException newException =
    new ArgumentOutOfRangeException(strMessage);
throw newException;
```

In this example, I first create a new instance of the `ArgumentOutOfRangeException` object and associate a custom error message with it, and then I throw the newly created exception.

You are not required to put this usage of the `throw` statement inside a `catch` block because you are just creating and throwing a new exception rather than rethrowing an existing one. You typically use this technique in raising your own custom exceptions. I discuss how to do that later in this chapter.

WARNING

Use `throw` Only When Required
The `throw` statement is an expensive operation. Use of `throw` consumes significant system resources compared to just returning a value from a method. You should use the `throw` statement cautiously and only when necessary because it has the potential to make your programs slow.

EXAM TIP

Custom Error Messages When you create an exception object, you should use its constructor that allows you to associate a custom error message rather than use its default constructor. The custom error message can pass specific information about the cause of the error and a possible way to resolve it.

Another way of throwing an exception is to throw it after wrapping it with additional useful information, as in this example:

```
catch(ArgumentNullException ane)
{
   //TODO: Add code to create an entry in the log file
   string strMessage = "CustomerID cannot be null";
   ArgumentNullException newException =
       new ArgumentNullException(strMessage, ane);
   throw newException;
}
```

Many times, you need to catch an exception that you cannot handle completely. In such a case you should perform any required processing and throw a more relevant and informative exception to the caller code so that it can perform the rest of the processing. In this case, you can create a new exception whose constructor wraps the previously caught exception in the new exception's `InnerException` property. The caller code then has more information available to handle the exception appropriately.

It is interesting to note that because `InnerException` is of type `Exception`, it also has an `InnerException` property that may store a reference to another exception object. Therefore, when you throw an exception that stores a reference to another exception in its `InnerException` property, you are actually propagating a chain of exceptions. This information is very valuable at the time of debugging and allows you to trace the path of a problem to its origin.

The `finally` Block

The `finally` block contains code that always executes, whether or not any exception occurs. You use the `finally` block to write cleanup code that maintains your application in a consistent state and preserves sanitation in the environment. For example, you can write code to close files, database connections, and related input/output resources in a `finally` block.

It is not necessary for a `try` block to have an associated `finally` block. However, if you do write a `finally` block, you cannot have more than one, and the `finally` block must appear after all the `catch` blocks.

Step by Step 3.3 illustrates the use of the `finally` block.

EXAM TIP

No Code in Between `try-catch-finally` Blocks When you write `try`, `catch`, and `finally` blocks, they should be in immediate succession of each other. You cannot write any other code between the blocks, although compilers allow you to place comments between them.

STEP BY STEP

3.3 Using the `finally` Block

1. Add a new Windows form to the project. Name it `StepByStep3_3`.
2. Place two `TextBox` controls (`txtFileName` and `txtText`), two `Label` controls (keep their default names), and a `Button` control (`btnSave`) on the form and arrange them as shown in Figure 3.5.
3. Attach the `Click` event handler to the `btnSave` control and add the following code to handle the `Click` event:

```
private void btnSave_Click(
    object sender, System.EventArgs e)
{
    // a StreamWriter writes characters to a stream
    StreamWriter sw = null;
    try
    {
        sw = new StreamWriter(txtFileName.Text);
        // Attempt to write the text box
        // contents in a file
        foreach(string line in txtText.Lines)
            sw.WriteLine(line);
        // This line only executes if there
        // were no exceptions so far
        MessageBox.Show(
          "Contents written, without any exceptions");
    }
    //catches all CLS-compliant exceptions
    catch(Exception ex)
    {
        string msg = String.Format(
            "Message: {0}\n Stack Trace:\n {1}",
            ex.Message, ex.StackTrace);
        MessageBox.Show(msg, ex.GetType().ToString());
        goto end;
    }
    // finally block is always executed to make sure
    // that the resources get closed whether or not
    // the exception occurs. Even if there is a goto
    // statement in catch or try block the final block
    // is first executed before the control goes to
    // the goto label
    finally
    {
        if (sw != null)
            sw.Close();
```

continues

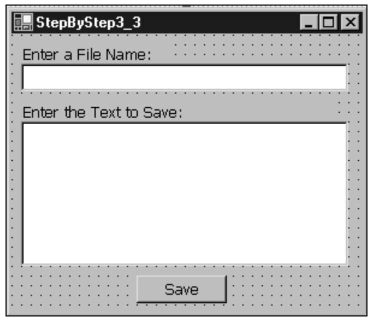


FIGURE 3.5
When you click the Save button, the code in `finally` block executes, regardless of any exception in the `try` block.

continued

```
            MessageBox.Show("Finally block always " +
              "executes whether or not exception occurs");
        }
    end:
        MessageBox.Show("Control is at label: end");
    }
```

4. Insert a `Main()` method to launch the form. Set the form as the startup object for the project.

5. Run the project. You should see a Windows form, as shown in Figure 3.5. Enter a filename and some text. Watch the order of messages. Note that the message box being displayed in the `finally` block is always displayed prior to the message box displayed by the `end` label.

EXAM TIP

The `finally` Block Always Executes If you have a `finally` block associated with a `try` block, the code in the `finally` block always executes, whether or not an exception occurs.

Step by Step 3.3 illustrates that the `finally` block always executes. In addition, if there is a transfer-control statement such as `goto`, `break`, or `continue` in either the `try` block or the `catch` block, the control transfer happens after the code in the `finally` block is executed. What happens if there is a transfer-control statement in the `finally` block also? That is not an issue because the C# compiler does not allow you to put a transfer-control statement such as `goto` inside a `finally` block.

One of the ways the `finally` statement can be used is in the form of a `try-finally` block without any `catch` block. Here is an example:

```
try
{
   //Write code to allocate some resources
}
finally
{
   //Write code to Dispose all allocated resources
}
```

NOTE

Throwing Exceptions from the `finally` Block Although it is perfectly legitimate to throw exceptions from a `finally` block, it is not recommended. The reason for this is that when you are processing a `finally` block, you might already have an unhandled exception waiting to be caught.

This use ensures that allocated resources are properly disposed of, no matter what. In fact, C# provides a `using` statement that does the exact same job but with less code. A typical use of the `using` statement is as follows:

```
// Write code to allocate some resource. List the
// allocate resources in a comma-separated list inside
// the parentheses, with the following using block
using(...)
```

```
{
  //use the allocated resource
}
// Here, the Dispose() method is called for all the
// objects referenced in the parentheses of the
// using statement. There is no need to write
// any additional code
```

R E V I E W   B R E A K

- An exception occurs when a program encounters any unexpected problem during normal execution.
- The FCL provides two main types of exceptions: `SystemException` and `ApplicationException`. `SystemException` represents the exceptions thrown by the CLR, and `ApplicationException` represents the exceptions thrown by the applications.
- The `System.Exception` class represents the base class for all CLS-compliant exceptions and provides the common functionality for exception handling.
- The `try` block consists of code that may raise an exception. A `try` block cannot exist on its own. It should be immediately followed by one or more `catch` blocks or a `finally` block.
- The `catch` block handles the exception raised by the code in the `try` block. The CLR looks for a matching `catch` block to handle the exception; this is the first `catch` block with either the exact same exception or any of the exception's base classes.
- If there are multiple `catch` blocks associated with a `try` block, the `catch` blocks should be arranged in specific-to-general order of exception types.
- The `throw` statement is used to raise an exception.
- The `finally` block is used to enclose code that needs to run, regardless of whether an exception is raised.

CREATING AND USING CUSTOM EXCEPTIONS

Implement error handling in the user interface.

- **Create and implement custom error messages.**
- **Create and implement custom error handlers.**
- **Raise and handle errors.**

EXAM TIP

Using `ApplicationException` as a Base Class for Custom Exceptions Although you can derive custom exception classes directly from the `Exception` class, Microsoft recommends that you derive custom exception classes from the `ApplicationException` class.

The exception classes provided by the .NET Framework, combined with the custom messages created when you create a new `Exception` object to throw or rethrow exceptions, should suffice for most of your exception handling requirements. In some cases, however, you might need exception types that are specific to the problem you are solving.

The .NET Framework allows you to define custom exception classes. To keep your custom-defined `Exception` class homogeneous with the .NET exception framework, Microsoft recommends that you consider the following when you design a custom exception class:

- Create an exception class only if there is no existing exception class that satisfies your requirement.
- Derive all programmer-defined exception classes from the `System.ApplicationException` class.
- End the name of your custom exception class with the word `Exception` (for example, `MyOwnCustomException`).
- Implement three constructors with the signatures shown in the following code:

```
public class MyOwnCustomException :
    ApplicationException
{
   // Default constructor
   public MyOwnCustomException ()
   {
   }
   // Constructor accepting a single string message
   public MyOwnCustomException (string message) :
       base(message)
   {
    }
    // Constructor accepting a string message and an
    // inner exception that will be wrapped
    // by this custom exception class
    public MyOwnCustomException(string message,
       Exception inner) : base(message, inner)
    {
    }
}
```

Step by Step 3.4 shows you how to create a custom exception.

STEP BY STEP

3.4 Creating and Using a Custom Exception

1. Add a new Windows form to the project. Name it `StepByStep3_4`.

2. Place and arrange controls on the form as shown in Figure 3.6. Name the `TextBox` control `txtDate`, the `Button` control `btnIsLeap`, and the `Label` control inside the Results panel `lblResult`.

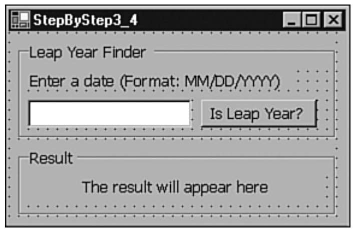

StepByStep3_4
Leap Year Finder
Enter a date (Format: MM/DD/YYYY)
Is Leap Year?
Result
The result will appear here

FIGURE 3.6
The leap year finder implements a custom exception for an invalid date format.

3. Switch to the code view and add the following definition for the `MyOwnInvalidDateFormatException` class to the end of the class definition for project `StepByStep3_4`:

```
// You can create your own exception classes by
// deriving from the ApplicationException class.
// It is good coding practice to end the class name
// of the custom exception with the word "Exception"
public class MyOwnInvalidDateFormatException :
    ApplicationException
{
    // It is a good practice to implement the three
    // recommended common constructors as shown here.
    public MyOwnInvalidDateFormatException()
    {
    }
    public MyOwnInvalidDateFormatException(
       string message): base(message)
    {
        this.HelpLink =
     "file://MyOwnInvalidDateFormatExceptionHelp.htm";
    }
    public MyOwnInvalidDateFormatException(
     string message, Exception inner) :
     base(message, inner)
    {
    }
}
```

continues

continued

4. Add the following definition for the `Date` class:

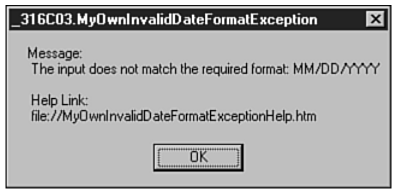
```
//This class does elementary date handling required
//for this program
public class Date
{
    private int day, month, year;

    public Date(string strDate)
    {
        if (strDate.Trim().Length == 10)
        {
            //Input data might be in an invalid format
            //In which case, Convert.ToDateTime()
            // method will fail
            try
            {
                DateTime dt =
                   Convert.ToDateTime(strDate);
                day = dt.Day;
                month = dt.Month;
                year = dt.Year;
            }
            //Catch the exception, attach that to the
            //custom exception and
            //throw the custom exception
            catch(Exception e)
            {
                throw new
                  MyOwnInvalidDateFormatException(
                  "Custom Exception Says: " +
                  "Invalid Date Format", e);
            }
        }
        else
            //Throw the custom exception
            throw new MyOwnInvalidDateFormatException(
               "The input does not match the " +
               "required format: MM/DD/YYYY");
    }

    //Find if the given date belongs to a leap year
    public bool IsLeapYear()
    {
        return (year%4==0) && ((year %100 !=0) ||
            (year %400 ==0));
    }
}
```

5. Add the following event handler for the `Click` event of `btnIsLeap`:

```
private void btnIsLeap_Click(
     object sender, System.EventArgs e)
{
    try
    {
        Date dt = new Date(txtDate.Text);
        if (dt.IsLeapYear())
            lblResult.Text =
               "This date is in a leap year";
        else
            lblResult.Text =
               "This date is NOT in a leap year";
    }
    //Catch the custom exception and
    //display an appropriate message
    catch (MyOwnInvalidDateFormatException dte)
    {
        string msg;
        //If some other exception was also
        //attached with this exception
        if (dte.InnerException != null)
          msg = String.Format(
          "Message:\n {0}\n\n Inner Exception:\n {1}",
          dte.Message, dte.InnerException.Message);
        else
            msg = String.Format(
               "Message:\n {0}\n\n Help Link:\n {1}",
               dte.Message, dte.HelpLink);

       MessageBox.Show(msg, dte.GetType().ToString());
    }
}
```

6. Insert a `Main()` method to launch the form. Set the form as the startup object for the project.

7. Run the project. Enter a date and click the button. If the date you enter is in the required format, you see a result displayed in the Results group box; otherwise, you get a message box showing the custom error message thrown by the custom exception, as in Figure 3.7.


_316C03.MyOwnInvalidDateFormatException

Message:
The input does not match the required format: MM/DD/YYYY

Help Link:
file://MyOwnInvalidDateFormatExceptionHelp.htm

FIGURE 3.7
You can associate a customized error message and a help link with a custom exception.

GUIDED PRACTICE EXERCISE 3.1

You are a Windows developer for a data analysis company. For one of your applications you need to create a keyword searching form that asks for a filename and a keyword from the user (as shown in Figure 3.8). The form should search for the keyword in the file and display the number of lines that contain the keyword in the results group box. Your form assumes that the entered keyword is a single word. If it is not a single word, you need to create and throw a custom exception for that case.

How would you throw a custom exception to implement custom error messages and custom error handling in your program?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

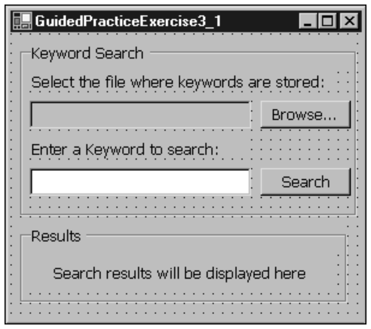

GuidedPracticeExercise3_1
Keyword Search
Select the file where keywords are stored:
Browse...
Enter a Keyword to search:
Search
Results
Search results will be displayed here

FIGURE 3.8
The keyword searching form throws a custom exception if the input is not in the required format.

1. Add a new form to your Visual C# .NET project. Name the form `GuidedPracticeExercise3_1.cs`.
2. Place and arrange controls on the form as shown in Figure 3.8. Name the `TextBox` control for accepting the filename `txtFileName` and the Browse control `btnBrowse`. Set the `ReadOnly` property of `txtFileName` to `true`. Name the `TextBox` control for accepting the keyword `txtKeyword` and the `Button` control `btnSearch`. Set the tab order of the form in the correct order, to ensure that the user's cursor is not placed in the read-only text box when the application starts.
3. Add an `OpenFileDialog` control to the form and change its name to `dlgOpenFile`.
4. Create a new class named `BadKeywordFormatException` that derives from `ApplicationException` and place the following code in it:

```
public class BadKeywordFormatException :
    ApplicationException
{
    public BadKeywordFormatException()
    {
    }
    public BadKeywordFormatException(string message):
        base(message)
    {
    }
    public BadKeywordFormatException(string message,
        Exception inner): base(message, inner)
    {
    }
}
```

5. Create a method named `GetKeywordFrequency()` in the `GuidedPracticeExercise3_1` class. This method should accept a string and return the number of lines that contain the string. Add the following code to the method:

```
private int GetKeywordFrequency(string path)
{
    if(this.txtKeyword.Text.Trim().IndexOf(' ') >= 0)
        throw new BadKeywordFormatException(
          "The keyword must only have a single word");

    int count = 0;
    if (File.Exists(path))
    {
        StreamReader sr =
           new StreamReader(txtFileName.Text);
        while (sr.Peek() > -1)
         if (sr.ReadLine().IndexOf(txtKeyword.Text)
           >= 0)
                count++;
    }
    return count;
}
```

6. Add the following code to the `Click` event handler of `btnBrowse`:

```
private void btnBrowse_Click(
    object sender, System.EventArgs e)
{
    if (dlgOpenFile.ShowDialog() == DialogResult.OK)
        txtFileName.Text = dlgOpenFile.FileName;
}
```

7. Add the following code to the `Click` event handler of `btnSearch`:

```
private void btnSearch_Click(
    object sender, System.EventArgs e)
{
    if (txtKeyword.Text.Trim().Length == 0)
    {
        MessageBox.Show(
            "Please enter a keyword to search for",
            "Missing Keyword");
        return;
    }
```

continues

continued

```
        try
        {
            lblResult.Text = String.Format(
             "The keyword: '{0}', was found in {1} lines",
              txtKeyword.Text,
              GetKeywordFrequency(txtFileName.Text));
        }
        catch(BadKeywordFormatException bkfe)
        {
            string msg = String.Format(
                "Message:\n {0}\n\n StackTrace:\n{1}",
                bkfe.Message, bkfe.StackTrace);
            MessageBox.Show(msg, bkfe.GetType().ToString());
        }
    }
```

8. Insert the `Main()` method to launch the form `GuidedPracticeExercise3_1.cs`. Set the form as the startup object for the project.

9. Run the project. Click the Browse button and select an existing file. Enter the keyword to search for in the file and click the Search button. If the keyword entered is in the wrong format (for example, if it contains two words), the custom exception is raised.

If you have difficulty following this exercise, review the sections "Handling Exceptions" and "Creating and Using Custom Exceptions" earlier in this chapter. After reviewing, try this exercise again.

MANAGING UNHANDLED EXCEPTIONS

The CLR-managed applications execute in an isolated environment called an *application domain*. The `AppDomain` class of the `System` namespace programmatically represents the application domain. The `AppDomain` class provides a set of events that allows you to respond when an assembly is loaded, when an application domain is unloaded, or when an application throws an unhandled exception.

06 0789728230 CH03 11/21/02 1:17 PM Page 251

In this chapter, we are particularly interested in the `UnhandledException` event of the `AppDomain` class, which occurs when any other exception handler does not catch an exception. Table 3.2 lists the properties of the `UnhandledExceptionEventArgs` class.

TABLE 3.2

IMPORTANT MEMBERS OF THE `UnhandledExceptionEventArgs` CLASS

Member	*Type*	*Description*
`ExceptionObject`	Property	Specifies the unhandled exception object that corresponds to the current domain
`IsTerminating`	Property	Indicates whether the CLR is terminating

You can attach an event handler with the `UnhandledException` event to take custom actions such as logging exception-related information. A log that is maintained over a period of time may help you find and analyze patterns with useful debugging information. There are several ways you can log information that is related to an event:

- By using the Windows event log
- By using custom log files
- By using databases such as SQL Server 2000
- By sending email notifications

Among these ways, the Windows event log offers the most robust method for event logging because it requires minimal assumptions for logging events. The other cases are not as fail-safe; for example, an application could loose connectivity with the database or with the SMTP server, or you might have problems writing an entry in a custom log file.

The .NET Framework provides you access to the Windows event log through the `EventLog` class. Windows 2000 and later have three default logs—application, system, and security. You can use the `EventLog` class to create custom event logs. You can easily view the event log by using the Windows Event Viewer utility.

You should familiarize yourself with the important members of the `EventLog` class that are listed in Table 3.3.

NOTE

When Not to Use the Windows Event Log The Windows event log is not available on older versions of Windows, such as Windows 98. If your application needs to support computers running older versions of Windows, you might want to create a custom error log. In a distributed application, you might want to log all events centrally in a SQL Server database. To make the scheme fail-safe, you can choose to log locally if the database is not available and transfer the log to the central database when it is available again.

TABLE 3.3

IMPORTANT MEMBERS OF THE EventLog CLASS

Member	*Type*	*Description*
`Clear()`	Method	Removes all entries from the event log and makes it empty
`CreateEventSource()`	Method	Creates an event source that you can use to write to a standard or custom event log
`Entries`	Property	Gets the contents of the event log
`Log`	Property	Specifies the name of the log to read from or write to
`MachineName`	Property	Specifies the name of the computer on which to read or write events
`Source`	Property	Specifies the event source name to register and use when writing to the event log
`SourceExists()`	Method	Specifies whether the event source exists on a computer
`WriteEntry()`	Method	Writes an entry in the event log

STEP BY STEP

3.5 Logging Unhandled Exceptions in the Windows Event Log

1. Add a new Windows form to the project. Name it `StepByStep3_5`.

2. Place three `TextBox` controls (`txtMiles`, `txtGallons`, and `txtEfficiency`) and a `Button` control (`btnCalculate`) on the form and arrange them as shown in Figure 3.1. Add `Label` controls as necessary.

3. Switch to the code view and add the following `using` directive at the top of the program:

```
using System.Diagnostics;
```

4. Double-click the `Button` control and add the following code to handle the `Click` event handler of the `Button` control:

06 0789728230 CH03 11/21/02 1:17 PM Page 253

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
    //This code has no error checking.
    //If something goes wrong at run time,
    //it will throw an exception
    decimal decMiles =
        Convert.ToDecimal(txtMiles.Text);
    decimal decGallons =
        Convert.ToDecimal(txtGallons.Text);
    decimal decEfficiency = decMiles/decGallons;
    txtEfficiency.Text =
        String.Format("{0:n}", decEfficiency);
}
```

5. Add the following code in the class definition:

```
private static void UnhandledExceptionHandler(
    object sender, UnhandledExceptionEventArgs ue)
{
    Exception unhandledException =
       (Exception) ue.ExceptionObject;

    //If no event source exist,
    //create an event source.
    if(!EventLog.SourceExists(
          "Mileage Efficiency Calculator"))
    {
        EventLog.CreateEventSource(
            "Mileage Efficiency Calculator",
            "Mileage Efficiency Calculator Log");
    }

    // Create an EventLog instance
    // and assign its source.
    EventLog eventLog = new EventLog();
    eventLog.Source = "Mileage Efficiency Calculator";

    // Write an informational entry to the event log.
    eventLog.WriteEntry(unhandledException.Message);
    MessageBox.Show("An exception occurred: " +
        "Created an entry in the log file");
}
```

6. Insert the following `Main()` method:

```
[STAThread]
public static void Main()
{
    // Create an AppDomain object
    AppDomain adCurrent = AppDomain.CurrentDomain;
    // Attach the UnhandledExceptionEventHandler to
    // the UnhandledException of the AppDomain object
    adCurrent.UnhandledException += new
```

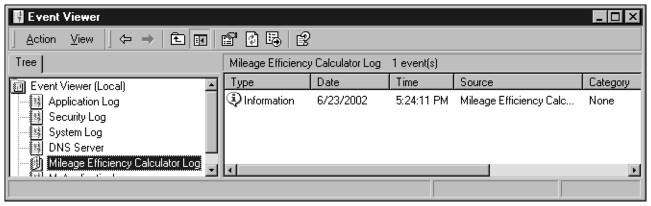
continues

continued

```
          UnhandledExceptionEventHandler(
          UnhandledExceptionHandler);
        Application.Run(new StepByStep3_5());
}
```

7. Set the form as the startup object for the project.

8. Run the project. Enter invalid values for miles and gallons and run the program. When an unhandled exception occurs, a message box is displayed, notifying you that the exception has been logged. You can view the logged message by selecting Event Viewer from the Administrative Tools section of the Control Panel. The Event Viewer displays the Mileage Efficiency Calculator Log and other logs in the left pane (see Figure 3.9). The right pane of the Event Viewer shows the events that are logged. You can double-click an event to view the description and other properties of the event, as shown in Figure 3.10.

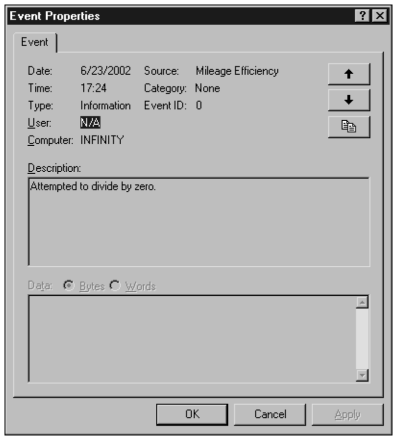


Event Viewer (Local)
Application Log
Security Log
System Log
DNS Server
Mileage Efficiency Calculator Log

Mileage Efficiency Calculator Log 1 event(s)

Type	Date	Time	Source	Category
Information	6/23/2002	5:24:11 PM	Mileage Efficiency Calc...	None

FIGURE 3.9
You can view messages logged to an event log by using the Windows Event Viewer.



Date: 6/23/2002 Source: Mileage Efficiency
Time: 17:24 Category: None
Type: Information Event ID: 0
User: N/A
Computer: INFINITY

Description:
Attempted to divide by zero.

FIGURE 3.10
You can view event properties for a particular event to see the event-related details.

REVIEW BREAK

- If the existing exception classes do not meet your exception handling requirements, you can create new exception classes that are specific to your application by deriving them from the `ApplicationException` class.
- You can use the `UnhandledException` event of the `AppDomain` class to manage unhandled exceptions.
- You can use the `EventLog` class to log events to the Windows event log.

VALIDATING USER INPUT

Validate user input.

Garbage in results in garbage out. When designing an application that accepts data from the user, you must ensure that the entered data is acceptable for the application. The most relevant place to ensure the validity of data is at the time of data entry itself. You can use various techniques for validating data:

- You can restrict the values that a field can accept by using standard controls such as combo boxes, list boxes, radio buttons, and check boxes. These allow users to select from a set of given values rather than permit free keyboard entry.
- You can capture the user's keystrokes and analyze them for validity. Some fields may require the user to enter only alphabetic values but no numeric values or special characters; in that case, you can accept the keystrokes for alphabetic characters while rejecting others.
- You can restrict entry in some data fields by enabling or disabling them, depending on the state of other fields.
- You can analyze the contents of the data field as a whole and warn the user of any incorrect values when he or she attempts to leave the field or close the window.

The first technique is discussed relative to the use of various controls in Chapter 2, "Controls"; the following sections cover rest of these techniques.

Keystroke-Level Validation

When you press a key on a control, three events take place, in the following order:

1. `KeyDown`
2. `KeyPress`
3. `KeyUp`

You can program the event handlers for these events in order to perform keystroke-level validation. You choose the event to program based on the order in which the event is fired and the information that is passed in the event argument of the event handler.

The `KeyPress` event happens after the `KeyDown` event but before the `KeyUp` event. Its event handler receives an argument of type `KeyPressEventArgs`. Table 3.4 lists the properties of `KeyPressEventArgs`.

TABLE 3.4

IMPORTANT MEMBERS OF THE KeyPressEventArgs CLASS

Member	*Type*	*Description*
`Handled`	Property	Indicates whether the event has been handled
`KeyChar`	Property	Returns the character value that corresponds to the key

The `KeyPress` event fires only if the key that is pressed generates a character value. To handle keypresses for function keys, control keys, and cursor movement keys, you must use the `KeyDown` and `KeyUp` events.

The `KeyDown` and `KeyUp` events occur when the user presses and releases a key on the keyboard, respectively. Event handlers of these events receive an argument of `KeyEventArgs` type; it provides the properties listed in Table 3.5.

TABLE 3.5

IMPORTANT MEMBERS OF THE KeyEventArgs CLASS

Member	*Type*	*Description*
`Alt`	Property	Returns `true` if the Alt key is pressed; otherwise, returns `false`.
`Control`	Property	Returns `true` if the Ctrl key is pressed; otherwise, returns `false`.
`Handled`	Property	Indicates whether the event has been handled.
`KeyCode`	Property	Returns the keyboard code for the event. Its value is one of the values specified in the `Keys` enumeration.
`KeyData`	Property	Returns the key code for the pressed key, along with modifier flags that indicate what combination of modifier keys (Ctrl, Shift, and Alt) are pressed at the same time.
`KeyValue`	Property	Returns the integer representation of the `KeyData` property.
`Modifiers`	Property	Returns the modifier flags that indicate what combination of modifier keys (Ctrl, Shift, and Alt) are pressed.
`Shift`	Property	Returns `true` if the Shift key is pressed; otherwise, returns `false`.

The `KeyPreview` Property

By default, only the active control receives the keystroke events. The `Form` object also has the `KeyPress`, `KeyUp`, and `KeyDown` events, but they are fired only when all the controls on the form are either hidden or disabled.

When you set the `KeyPreview` property of a form to `true`, the form receives all three events—`KeyPress`, `KeyUp`, and `KeyDown`—just before the active control receives these events. This allows you to set up a two-tier validation on controls. If you want to discard certain types of characters at the form level, you can set the `Handled` property for the event argument to `true` (this does not allow the event to propagate to the active control); otherwise, the events propagate to the active control. You can then use keystroke events at the control level to perform field-specific validations, such as restricting the field to only numeric digits.

Field-Level Validation

Field-level validation ensures that the value entered in the field is in accordance with the application's requirements. If it is not, you can display an error to alert the user about the problem. These are appropriate reasons to perform field-level validations:

- When the user attempts to leave the field
- When the content of the field changes for any reason

When the user enters and leaves a field, the events occur in the following order:

1. `Enter` (Occurs when a control is entered.)
2. `GotFocus` (Occurs when a control receives focus.)
3. `Leave` (Occurs when focus leaves a control.)
4. `Validating` (Occurs when a control is validating.)
5. `Validated` (Occurs when a control is finished validating.)
6. `LostFocus` (Occurs when a control looses focus.)

The `Validating` event is the ideal place to store the validating logic for a field. The following sections explain the use of the `Validating` event and the `CausesValidation` property for field-level validation. They also discuss the use of the `ErrorProvider` component to display error messages to the user.

The `Validating` Event

The `Validating` event is the ideal place for storing the field-level validation logic for a control. The event handler for validating the event receives an argument of type `CancelEventArgs`. Its only property, `Cancel`, cancels the event when it is set to `true`.

Inside the `Validating` event, you can write code to do the following:

- Programmatically correct any errors or omissions made by the user.
- Show error messages and alerts to the user so that the user can fix the problem.

Inside the `Validating` event, you might also want to retain the focus in the current control, thus forcing the user to fix the problem before proceeding further. To do this, you can use either of the following techniques:

- Use the `Focus()` method of the control to transfer the focus back to the field.
- Set the `Cancel` property of `CancelEventArgs` to `true`. This cancels the `Validating` event, leaving the focus in the control.

A related event, `Validated`, is fired just after the `Validating` event occurs—and it enables you to take actions after the control's contents have been validated.

NOTE

The `Validating` Event and Sticky Forms The `Validating` event fires when you close a form. If inside the `Validating` event you set the `Cancel` property of the `CancelEventArgs` argument to `true`, the `Validating` event also cancels the close operation.

There is a workaround for this problem. Inside the `Validating` event, you should set the `Cancel` property of the `CancelEventArgs` argument to `true` if the mouse is in the form's client area. The close button is in the title bar that is outside the client area of the form. Therefore, when the user clicks the close button, the `Cancel` property is not set to `true`.

The `CausesValidation` Property

When you use the `Validating` event to restrict the focus in the control by canceling the event, you must also consider that you are making the control sticky.

Consider a case in which the user is currently on a control such as a `TextBox` control, with incorrect data, and you are forcing the user to fix the problem before leaving the control, by setting the `Cancel` property of `CancelEventArgs` to `true`. When the user clicks the Help button in the toolbar to check what is wrong, nothing happens unless the user makes a correct entry. This can be an annoying situation for the user, so you want to avoid it in your applications.

The `CausesValidation` property comes to your rescue in such a case. The default value of the `CausesValidation` property for a control is `true` for all controls, which means that the `Validating` event fires for any control, requiring validation before the control in question receives the focus.

When you want a control to respond, regardless of the validation status of other controls, you should set the `CausesValidation` property of that control to `false`. For example, in the previous example, the Help button in the toolbar would be set with the `CausesValidation` property set to `false`.

The `ErrorProvider` Component

The `ErrorProvider` component in the Visual Studio .NET toolbox is useful when you're showing validation-related error messages to the user. The `ErrorProvider` component can set a small icon next to a field when it contains an error. When the user moves the mouse pointer over the icon, an error message pops up as a ToolTip. This is a better way of displaying error messages than the old way of using message boxes because it eliminates at least two serious problems with message boxes:

- When you use message boxes, if you have errors on multiple controls, popping up several message boxes might annoy or scare your users.
- After the user dismisses a message box, the error message is no longer available for reference.

Table 3.6 lists some important members of the `ErrorProvider` class with which you should familiarize yourself.

TABLE 3.6

IMPORTANT MEMBERS OF THE `ErrorProvider` CLASS

Member	*Type*	*Description*
`BlinkRate`	Property	Specifies the rate at which the error icon flash-es.
`BlinkStyle`	Property	Specifies a value that indicates when the error icon flashes.
`ContainerControl`	Property	Specifies the component's parent control.
`GetError()`	Method	Returns the error description string for the specified control.
`Icon`	Property	Specifies an icon to display next to a control. The icon is displayed only when an error description string has been set for the control.
`SetError()`	Method	Sets the error description string for the speci-fied control.
`SetIconAlignment()`	Method	Sets the location at which to place an error icon with respect to the control. It has one of the `ErrorIconAlignment` values (`BottomLeft`, `BottomRight`, `MiddleLeft`, `MiddleRight`, `TopLeft`, and `TopRight`).
`SetIconPadding()`	Method	Specifies the amount of extra space to leave between the control and the error icon.

The `ErrorProvider` component displays an error icon next to a field, based on the error message string. The error message string is set by the `SetError()` method. If the error message is empty, no error icon is displayed, and the field is considered correct. Step by Step 3.5 shows how to use the `ErrorProvider` component.

STEP BY STEP

3.6 Using the `ErrorProvider` Component and Other Validation Techniques

1. Add a new Windows form to the project. Name it `StepByStep3_6`.

2. Place three `TextBox` controls (`txtMiles`, `txtGallons`, and `txtEfficiency`) and a `Button` control (`btnCalculate`) on the form and arrange them as shown in Figure 3.11. Add `Label` controls as necessary.

3. The `ErrorProvider` component is present in the Windows Forms tab of the Visual Studio .NET toolbox. Add an `ErrorProvider` component (`errorProvider1`) to the form. The `ErrorProvider` component is placed in the component tray.

4. Double-click the form and add the following code to handle the `Load` event handler of the `Form` control:

```
private void StepByStep3_6_Load(
    object sender, System.EventArgs e)
{
    // Set the ErrorProvider's Icon
    // alignment for the TextBox controls
    errorProvider1.SetIconAlignment(
        txtMiles, ErrorIconAlignment.MiddleLeft);
    errorProvider1.SetIconAlignment(
        txtGallons, ErrorIconAlignment.MiddleLeft);
}
```

5. Attach the `Validating` event handlers to the `TextBox` controls and add the following code to handle the `Validating` event handler of the `txtMiles` and `txtGallons` controls:

```
private void txtMiles_Validating(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    try
```

continues

continued

```
    {
        decimal decMiles =
           Convert.ToDecimal(txtMiles.Text);
        errorProvider1.SetError(txtMiles, "");
    }
    catch(Exception ex)
    {
        errorProvider1.SetError(txtMiles, ex.Message);
    }
}

private void txtGallons_Validating(object sender,
     System.ComponentModel.CancelEventArgs e)
{
    try
    {
        decimal decGallons =
             Convert.ToDecimal(txtGallons.Text);
        if (decGallons > 0)
            errorProvider1.SetError(txtGallons, "");
        else
            errorProvider1.SetError(txtGallons,
             "Please enter a value > 0");
    }
    catch(Exception ex)
    {
        errorProvider1.SetError(
           txtGallons, ex.Message);
    }
}
```

6. Add the following code to the `Click` event handler of `btnCalculate`:

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
   // Check whether the error description is not empty
   // for either of the TextBox controls
   if (errorProvider1.GetError(txtMiles) != "" ||
       errorProvider1.GetError(txtGallons) != "")
       return;

    try
    {
        decimal decMiles =
            Convert.ToDecimal(txtMiles.Text);
        decimal decGallons =
            Convert.ToDecimal(txtGallons.Text);
        decimal decEfficiency = decMiles/decGallons;
        txtEfficiency.Text =
            String.Format("{0:n}", decEfficiency);
    }
    catch(Exception ex)
    {
        string msg = String.Format(
            "Message: {0}\n Stack Trace:\n {1}",
            ex.Message, ex.StackTrace);
        MessageBox.Show(msg, ex.GetType().ToString());
    }
}
```

7. Insert the `Main()` method to launch the form. Set the form as the startup object for the project.

8. Run the project. Enter values for miles and gallons and click Calculate. The program calculates the mileage efficiency, as expected. When you enter an invalid value into any of the `TextBox` controls, the error icon starts blinking and displays the error message when the mouse is hovered over the error icon, as shown in Figure 3.11.

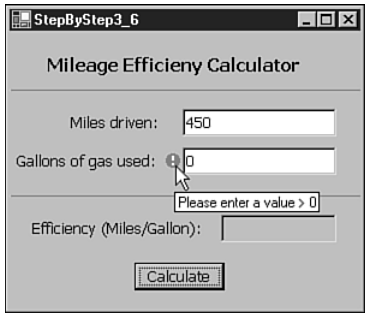


FIGURE 3.11
The `ErrorProvider` component shows the error icon and the error message in a nonintrusive way.

Enabling Controls Based on Input

One of the useful techniques for restricting user input is selectively enabling and disabling controls. These are some common cases in which you would want to do this:

- Your application might have a check box titled Check Here if You Want to Ship to a Different Location. Only when the user checks the check box should you allow him or her to enter values in the fields for the shipping address. Otherwise, the shipping address is the same as the billing address.
- In a Find dialog box, you have two buttons: Find and Cancel. You want to keep the Find button disabled initially and enable it only when the user enters search text in a text box.

The `Enabled` property for a control is `true` by default. When you set it to `false`, the control cannot receive the focus and appears grayed out.

For a control such as `TextBox`, you can also use the `ReadOnly` property to restrict user input. One advantage of using the `ReadOnly` property is that the control is still able to receive focus, so you are able to scroll through any text in the control that is not initially visible.

In addition, you can select and copy the text to the Clipboard, even if the `ReadOnly` property is `true`.

Other Properties for Validation

In addition to the techniques mentioned in the preceding sections, the properties described in the following sections allow you to enforce some restrictions on user input.

The `CharacterCasing` Property

The `CharacterCasing` property of the `TextBox` control changes the case of characters in the text box as required by the application. For example, you might want to convert all characters entered in a text box used for entering a password to lowercase so that there are no problems due to case-sensitivity.

The values of the `CharacterCasing` property can be set to three values: `CharacterCasing.Lower`, `CharacterCasing.Normal` (the default value), and `CharacterCasing.Upper`.

The `MaxLength` Property

The `MaxLength` property of a `TextBox` or `ComboBox` control specifies the maximum number of characters that the user can enter into the control. This property is handy when you want to restrict the size of some fields, such as fields for telephone numbers or zip codes. This property is useful in scenarios in which you are adding or updating records in a database with the values entered in the controls; in such a case you can use the `MaxLength` property to prevent the user from entering more characters than the corresponding database field can handle.

When the `MaxLength` property is zero (the default), the number of characters that can be entered is limited only by the available memory.

EXAM TIP

The Scope of the `MaxLength` Property The `MaxLength` property affects only the text that is entered into the control interactively by the user. Programmatically, you can set the value of the `Text` property to a value that is longer than the value specified by the `MaxLength` property.

GUIDED PRACTICE EXERCISE 3.2

As a Windows developer for a data analysis company, you recently developed a keyword searching form for your Windows application (refer to Guided Practice Exercise 3.1). The form asks for a filename and a keyword from the user, and then it searches for the keyword in the file and displays the number of lines that contain the keyword in the results group box. The form assumes that the entered keyword is a single word. In the solution in Guided Practice Exercise 3.1, if the keyword is not a single word, the form creates and throws a custom exception for that case. Since you created that solution, you have studied field-level validation techniques and realized that for this scenario, the use of field-level validation provides a much more elegant solution.

You now want to modify the keyword searching form. Its basic functionality is still the same as in Guided Practice Exercise 3.1, but you need to incorporate a few changes in the user interface. Initially the keyword text box and the Search button are disabled; you should enable these controls as the user progresses through the application. If the keyword entered by the user is not a single word, instead of throwing an exception, you need to display the error icon with the keyword text box and set an error message. The keyword text box should not lose focus unless it has valid data.

How would you create such a form?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Add a new form to your Visual C# .NET project. Name the form `GuidedPracticeExercise3_2.cs`.
2. Place and arrange the controls on the form as shown in Figure 3.8. Name the `TextBox` control for accepting the filename `txtFileName` and the `Browse` button `btnBrowse`. Set the `ReadOnly` property of `txtFileName` to `true`. Name the `TextBox` control for accepting the keyword `txtKeyword` and the `Button` control `btnSearch`. Set the tab order of the form in the correct order so that the user's cursor is not placed in a read-only text box when the application starts.

continues

continued

3. Add an `OpenFileDialog` control to the form and change its name to `dlgOpenFile`. Add an `ErrorProvider` component (`errorProvider1`) to the form. The `ErrorProvider` component is placed in the component tray.

4. Double-click the form to attach the `Load` event handler to the form. Add the following code to handle the `Load` event of the `Form` control:

```
private void GuidedPracticeExercise3_2_Load(
    object sender, System.EventArgs e)
{
    // Disable the keyword text box and Search button
    txtKeyword.Enabled = false;
    btnSearch.Enabled = false;
    errorProvider1.SetIconAlignment(
        txtKeyword, ErrorIconAlignment.MiddleLeft);
}
```

5. Attach `TextChanged` and `Validating` event handlers to the `txtKeyword` control and add the following code:

```
private void txtKeyword_TextChanged(
    object sender, System.EventArgs e)
{
    if(this.txtKeyword.Text.Length==0)
        this.btnSearch.Enabled = false;
    else
        this.btnSearch.Enabled = true;
}

private void txtKeyword_Validating(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    if(this.txtKeyword.Text.Trim().IndexOf(' ') >= 0)
    {
        errorProvider1.SetError(txtKeyword,
          "You must only specify a single word");
        txtKeyword.Focus();
        txtKeyword.Select(0, txtKeyword.Text.Length);
    }
    else
        errorProvider1.SetError(txtKeyword, "");
}
```

6. Create a method named `GetKeywordFrequency()` that accepts a string and returns the number of lines containing it. Add the following code to the method:

```
private int GetKeywordFrequency(string path)
{
    int count = 0;
    if (File.Exists(path))
    {
        StreamReader sr =
            new StreamReader(txtFileName.Text);
        while (sr.Peek() > -1)
            if (sr.ReadLine().IndexOf(txtKeyword.Text)
                >= 0)
                count++;
    }
    return count;
}
```

7. Add the following code to the `Click` event handler of `btnBrowse`:

```
private void btnBrowse_Click(
    object sender, System.EventArgs e)
{
    if (dlgOpenFile.ShowDialog() == DialogResult.OK)
    {
        txtFileName.Text = dlgOpenFile.FileName;
        this.txtKeyword.Enabled = true;
        this.txtKeyword.Focus();
    }
}
```

8. Add the following code to the `Click` event handler of `btnSearch`:

```
private void btnSearch_Click(
    object sender, System.EventArgs e)
{
    if (errorProvider1.GetError(txtKeyword) != "")
        return;
    try
    {
        lblResult.Text = String.Format(
          "The keyword: '{0}' was found in {1} lines",
          txtKeyword.Text,
          GetKeywordFrequency(txtFileName.Text));
    }
    catch(Exception ex)
    {
        string msg = String.Format(
            "Message:\n {0}\n\n StackTrace:\n{1}",
            ex.Message, ex.StackTrace);
        MessageBox.Show(msg, ex.GetType().ToString());
    }
}
```

9. Insert the `Main()` method to launch form `GuidedPracticeExercise3_2.cs`. Set the form as the startup object for the project.

continues

continued

10. Run the project. The keyword text box and the search button are disabled. Click the Browse button and select an existing file; this enables the keyword text box. Enter the keyword to search in the file; this enables the Search button. Click the Search button. If the keyword entered is in the wrong format (for example, if it contains two words), the `ErrorProvider` component shows the error message and the icon.

If you have difficulty following this exercise, review the section "Validating User Input," earlier in this chapter, and then try this exercise again.

REVIEW BREAK

- It a good practice to validate user input at the time of data entry. Thoroughly validated data results in consistent and correct data stored by the application.
- When a user presses a key, three events are generated, in the following order: `KeyDown`, `KeyPress`, and `KeyUp`.
- The `Validating` event is the ideal place for storing the field-level validation logic for a control.
- The `CausesValidation` property specifies whether validation should be performed. If it is set to `false`, the `Validating` and `Validated` events are suppressed.
- The `ErrorProvider` component in the Visual Studio .NET toolbox is used to show validation-related error messages to the user.
- A control cannot receive the focus and appears grayed out if its `Enabled` property is set to `false`.

CHAPTER SUMMARY

The .NET Framework provides fully integrated support for exception handling. In fact, it allows you to raise exceptions in one language and catch them in a program written in another language. The `try` block is used to enclose code that might cause exceptions. The `catch` block is used to handle the exceptions raised by the code in the `try` block, and the `finally` block ensures that certain code is executed, regardless of whether an exception occurs.

The FCL provides a large number of exception classes that represent most of the exceptions that a program may encounter. If you prefer to create your own custom exception class, you can do so by deriving your exception class from the `ApplicationException` class.

This chapter describes a variety of ways to validate user input. The Windows Forms library provides an `ErrorProvider` component that is used to signal errors. You can also associate custom icons and error messages with the `ErrorProvider` component.

KEY TERMS

- exception
- exception handling
- input validation

APPLY YOUR KNOWLEDGE

Exercises

3.1 Handling Exceptions

Recall that Step by Step 2.5 in Chapter 2 demonstrates the use of common dialog boxes through the creation of a simple rich text editor. This editor allows you to open and save a rich text file. You can also edit the text and change its fonts and colors. The program works fine in all cases except when you try to open or save a file that is already open; in that case, the program throws a `System.IO.IOException` exception.

The objective of this exercise is to make a robust version of this program that generates a warning about the open file rather than abruptly terminating the program.

Estimated time: 15 minutes

1. Open a new Windows application in Visual C# .NET. Name it `316C03Exercises`.
2. Add a Windows form to the project. Name the form `Exercise3_1`.
3. Place five `Button` controls on the form. Name them `btnOpen`, `btnSave`, `btnClose`, `btnColor`, and `btnFont`, and change their `Text` properties to `&Open...`, `&Save...`, `Clos&e...`, `&Color...`, and `&Font...`, respectively. Place a `RichTextBox` control on the form and name it `rtbText`. Arrange all the controls as shown in Figure 3.12.

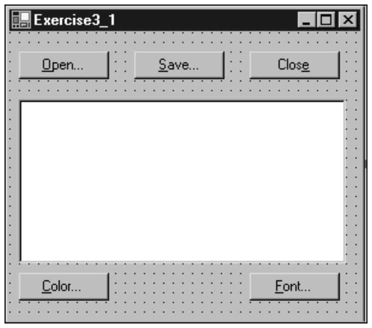


FIGURE 3.12
This robust version of a simple rich text editor handles the exceptions of `System.IO.IOException` type.

4. Drag and drop the following components from the toolbox onto the form: `OpenFileDialog`, `SaveFileDialog`, `ColorDialog`, and `FontDialog`.
5. Switch to the code view and add the following `using` directive to the top of the program:

   ```
   using System.IO;
   ```

6. Double-click the Open button to attach an event handler to this `Click` event. Add the following code to the event handler:

   ```
   private void btnOpen_Click(
       object sender, System.EventArgs e)
   {
       //Allow user to select only *.rtf files
       openFileDialog1.Filter =
           "Rich Text Files (*.rtf)|*.rtf";
       if(openFileDialog1.ShowDialog()
           == DialogResult.OK)
       {
           try
           {
              //Load the file contents
              //into the RichTextBox control
              rtbText.LoadFile(
              openFileDialog1.FileName,
              RichTextBoxStreamType.RichText);
           }
           catch(System.IO.IOException ioe)
   ```

APPLY YOUR KNOWLEDGE

```
        {
            MessageBox.Show(ioe.Message,
               "Error opening file");
        }
    }
}
```

7. Add the following code to handle the `Click` event of the Save button:

```
private void btnSave_Click(
    object sender, System.EventArgs e)
{
    //Default choice for saving file
    //is *.rtf but user can select
    //All Files to save with
    //another extension
    saveFileDialog1.Filter =
     "Rich Text Files (*.rtf)|*.rtf|" +
     "All Files (*.*)|*.*";
    if(saveFileDialog1.ShowDialog()
        == DialogResult.OK)
    {
        try
        {
           //Save the RichTextBox control's
           //content to a file
           rtbText.SaveFile(
           saveFileDialog1.FileName,
           RichTextBoxStreamType.RichText);
        }
        catch(System.IO.IOException ioe)
        {
            MessageBox.Show(ioe.Message,
               "Error saving file");
        }
    }
}
```

8. Add the following code to handle the `Click` event of the Close button:

```
private void btnClose_Click(
    object sender, System.EventArgs e)
{
    //close the form
    this.Close();
}
```

9. Add the following code to handle the `Click` event of the Color button:

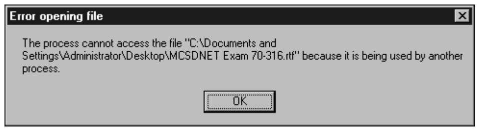
```
private void btnColor_Click(
    object sender, System.EventArgs e)
{
    if(colorDialog1.ShowDialog()
        == DialogResult.OK)
    {
        //Change the color of the selected
        //text. If no text is selected,
        // change the active color
        rtbText.SelectionColor =
            colorDialog1.Color;
    }
}
```

10. Add the following code to handle the `Click` event of the Font button:

```
private void btnFont_Click(
    object sender, System.EventArgs e)
{
    if(fontDialog1.ShowDialog()
       == DialogResult.OK)
    {
        //Change the font of the selected
        //text. If no text is selected,
        //change the active font
        rtbText.SelectionFont =
            fontDialog1.Font;
    }
}
```

11. Insert the `Main()` method to launch the form. Set the form as the startup object.

12. Run the project. Click on the Open button and try to open an already opened file. An error message appears, warning about the file already being open, as shown in Figure 3.13.



The process cannot access the file "C:\Documents and Settings\Administrator\Desktop\MCSDNET Exam 70-316.rtf" because it is being used by another process.

FIGURE 3.13
Instead of abnormal program termination, you now get an error message about the already open file.

APPLY YOUR KNOWLEDGE

3.2 Validating User Input

One technique for input validation is to force the user to fix an erroneous field before allowing him or her to move to another field. To achieve this, you can set the `Cancel` property of the `CancelEventArgs` argument of the field's `Validating` event to `false`.

In this exercise, you create a login form (see Figure 3.14) that accepts a username and password. It forces the user to enter the username. The user should also be able to close the application by clicking the Cancel button, regardless of the validation status of the fields.

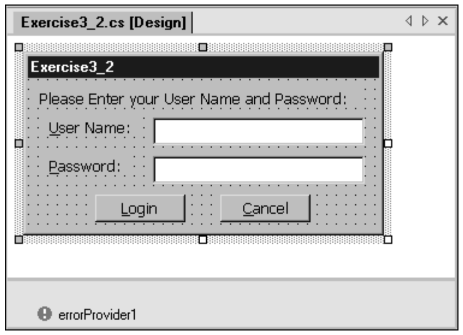


FIGURE 3.14
A nonsticky login form validates the input and allows users to close the application by clicking the Cancel button.

Estimated time: 15 minutes

1. Open a Visual C# .NET Windows application in the Visual Studio .NET IDE. Name it `316C03Exercises`.

2. Add a new form to the application. Name it `Exercise3_2`.

3. Place three `Label` controls (keep their default names), two `TextBox` controls (`txtUserName` and `txtPassword`), two `Button` controls (`btnLogin` and `btnCancel`), and an `ErrorProvider` component (`errorProvider1`) on the form. The `ErrorProvider` component is placed in the component tray. Arrange the controls in the form as shown in Figure 3.14.

4. Change the `ControlBox` property of the form to `false`, the `CharacterCasing` property of the `txtPassword` control to `Lower`, and the `CausesValidation` property of the `btnCancel` control to `false`.

5. Double-click the `Form` control to attach a `Load` event handler; add the following code to the event handler:

```
private void Exercise3_2_Load(
    object sender, System.EventArgs e)
{
    errorProvider1.SetIconAlignment(
         txtUserName,
         ErrorIconAlignment.MiddleLeft);
    errorProvider1.SetIconAlignment(
         txtPassword,
         ErrorIconAlignment.MiddleLeft);
}
```

6. Declare the following variable outside a method block in the class:

```
//closingFlag is used to check if the
//user has clicked the Close button
private bool closingFlag = false;
```

7. Add the following code to the `Click` event handler of the Cancel button:

```
private void btnCancel_Click(
    object sender, System.EventArgs e)
{
    closingFlag = true;
    this.Close();
}
```

APPLY YOUR KNOWLEDGE

8. Add the following code to the `Click` event handler of the Login button:

```
private void btnLogin_Click(
    object sender, System.EventArgs e)
{
    string strMessage = String.Format(
     "The following information:" +
     "\n\nUserName: {0}\n\nPassword: {1}" +
     "\n\n can now be passed to the " +
     "middle-tier for validation",
     txtUserName.Text, txtPassword.Text);
    MessageBox.Show(strMessage,
        "User Input Validation Succeeded");
}
```

9. Attach the following event handling code to the `Validating` events of both the `txtUserName` and `txtPassword` controls:

```
private void
   txtUserNamePassword_Validating(
   object sender,
   System.ComponentModel.CancelEventArgs e)
{
    TextBox fieldToValidate =
        (TextBox) sender;

    if (!closingFlag)
    {
    if(fieldToValidate.Text.Trim().Length
        == 0)
        {
            errorProvider1.SetError(
              fieldToValidate,
     "Please enter a value for this field");
            e.Cancel = true;
        }
        else if (
   fieldToValidate.Text.Trim().IndexOf(' ')
       >=0)
        {
            errorProvider1.SetError(
              fieldToValidate,
  "You may NOT have spaces in this field");
            fieldToValidate.Select(0,
              fieldToValidate.Text.Length);
            e.Cancel = true;
        }
    }
}
```

10. Attach the following event handling code to the `Validated` event of both the `txtUserName` and `txtPassword` controls:

```
private void txtUserNamePassword_Validated(
    object sender, System.EventArgs e)
{
    TextBox fieldToValidate =
       (TextBox) sender;
    errorProvider1.SetError(
       fieldToValidate, "");
}
```

11. Insert the `Main()` method to launch the form. Set the form as the startup object.

12. Run the project. Click the Login button, and you are forced to enter the username. However, you can click the Cancel button to close the application.

Review Questions

1. What is the default behavior of the .NET Framework when an exception is raised?
2. What is the base class of all exceptions that provides basic functionality for exception handling? What are the two main types of exception classes and their purposes?
3. Explain the `Message` and `InnerException` properties of the `Exception` class.
4. What is the purpose of a `try-catch` block?
5. How many `catch` blocks can be associated with a `try` block? How should they be arranged?
6. What is the importance of a `finally` block?
7. Can you associate custom error messages with the exception types defined by the CLR? If yes, how do you do it?

PART II

TESTING, DEBUGGING, AND DEPLOYING A WINDOWS APPLICATION

OBJECTIVES

This chapter covers the following Microsoft-specified objective for the "Testing and Debugging" section of Exam 70-316, "Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET":

Create a unit test plan.

▶ Before you release a product or component, the product needs to pass through different types of tests. This objective requires you to know the different types of tests that a product should undergo to verify its robustness, reliability, and correctness. These tests should be executed with a designed test plan that ensures that the product thoroughly meets its goals and requirements.

Implement tracing.

- **Add trace listeners and trace switches to an application.**
- **Display trace output.**

▶ Tracing helps in displaying informative messages during the application's runtime to get a fair idea of how the application is progressing. This objective requires you to know how to use `Trace` class properties and methods, attach trace listeners, and apply trace switches. Trace switches allow you to enable, disable, and filter tracing output that is displayed by the `Trace` class without recompiling programs. You can do this by just editing the configuration XML file.

CHAPTER 12

Testing and Debugging a Windows Application

OBJECTIVES

Debug, rework, and resolve defects in code.

- **Configure the debugging environment.**
- **Create and apply debugging code to components and applications.**
- **Provide multicultural test data to components and applications.**
- **Execute tests.**
- **Resolve errors and rework code.**

▶ The process of debugging helps you locate logical or runtime errors in an application. This objective requires you to know the various tools and windows that are available in Visual C# .NET to enable easy and effective debugging. These debugging tools and windows help a great deal in determining errors, executing test code, and resolving errors.

OUTLINE

STUDY STRATEGIES

- Review the “Introduction to Instrumentation and Tracing” and “Using the Debugger” sections of the Visual Studio .NET Combined Help Collection.
- Try calling different methods of the `Trace` and `Debug` classes. Note the differences in the output when you run a program using the `Debug` and `Release` configurations.
- Experiment with attaching predefined and custom-made listeners to `Trace` objects. Refer to Step by Step 12.2 and Guided Practice Exercise 12.1 for examples.
- Know how to implement trace switches and conditional compilation in Windows applications. Refer to Step by Step 12.3 and Step by Step 12.4 for examples.
- Experiment with the different types of debugging windows that are available in Visual C# .NET. Understand their advantages and learn to use them effectively. They can be very helpful in resolving errors.
- Experiment with various techniques for debugging, such as local and remote debugging, debugging code in DLLs, and debugging SQL Server stored procedures.

INTRODUCTION

Building a quality Windows application requires thorough testing to ensure that the application has the fewest possible defects. Therefore, you need to chart an effective test plan. Complex applications require multiple levels of testing, including unit testing, integration testing, and regression testing.

Tracing is the process of monitoring an executing program. You trace a program by placing tracing code in the program with the help of the `Trace` and `Debug` classes. The tracing messages can be sent to a variety of destinations, including the Output window, a text file, an event log, or any other custom-defined trace listener, where they can be recorded to analyze the behavior of the program. Trace switches can be used to change the types of messages being generated without recompiling the application.

The process of testing may reveal various logical errors, or bugs, in a program. The process of finding the exact locations of these errors may be time-consuming. Visual C# .NET provides a rich set of debugging tools that makes this process very convenient.

In this chapter I first discuss the test plan and various common testing techniques. I then discuss how to put tracing code in a program to monitor its execution. Finally, I talk about the debugging capabilities of Visual Studio .NET.

TESTING

NOTE

Correctness, Robustness, and Reliability *Correctness* refers to the ability of a program to produce expected results when the program is given a set of valid input data. *Robustness* is the ability of a program to cope up with invalid data or operations. *Reliability* is the ability of a program to produce consistent results on every use.

Testing is the process of executing a program with the intention of finding errors (bugs). By *error* I mean any case in which a program's actual results fail to match the expected results. The criteria of the expected results may not include just the correctness of the program; they may also include other attributes, such as usability, reliability, and robustness. The process of testing may be manual, automated, or a mixture of both techniques.

In this increasingly competitive world, testing is more important than ever. A software company cannot afford to ignore the importance of testing. If a company releases buggy code, not only will it end up spending more time and money fixing and redistributing the corrected code, but it will also lose goodwill. In the Internet world, the competition is not even next door: It is just a click away!

Creating a Test Plan

Create a unit test plan.

A *test plan* is a document that guides the process of testing. A good test plan should typically include the following information:

- Which software components needs to be tested
- What parts of a component's specification are to be tested
- What parts of a component's specification are not to be tested
- What approach needs to be followed for testing
- Who will be responsible for each task in the testing process
- What the schedule is for testing
- What the criteria are for a test to fail or pass
- How the test results will be documented and used

Executing Tests

Debug, rework, and resolve defects in code.

- **Execute tests.**

Incremental testing (sometime also called *evolutionary testing*) is a modern approach to testing that has proven very useful for rapid application development (RAD). The idea of incremental testing is to test the system as you build it. Three levels of testing are involved in incremental testing:

- **Unit testing**—Unit testing involves testing elementary units of the application (usually classes).
- **Integration testing**—Integration testing tests the integration of two or more units or the integration between subsystems of those units.
- **Regression testing**—Regression testing usually involves the process of repeating the unit and integration tests whenever a bug is fixed, to ensure that the old bugs do not exist and that no new ones have been introduced.

Unit Testing

Units are the smallest building blocks of an application. In Visual C# .NET, these building blocks often refer to components or class definitions. Unit testing involves performing basic tests at the component level to ensure that each unique path in the component behaves exactly as documented in its specifications.

NOTE

NUnit `NUnit` is a simple framework for writing repeatable tests in any .NET language. For more information, visit `http://nunit.sourceforge.net`.

Usually, the same person who writes the component also does unit testing for it. Unit testing typically requires that you write special programs that use the component or class being tested. These programs are called *test drivers*; they are used throughout the testing process, but they are not part of the final product.

The following are some of the major benefits of unit testing:

- It allows you to test parts of an application without waiting for the other parts to be available.
- It allows you to test exceptional conditions that are not easily reached by external inputs in a large, integrated system.
- It simplifies the debugging process by limiting the search for bugs to a small unit rather than to the complete application.
- It helps you avoid lengthy compile-build-debug cycles when debugging difficult problems.
- It enables you to detect and remove defects at a much lower cost than with other, later, stages of testing.

Integration Testing

Integration testing verifies that the major subsystems of an application work well with each other. The objective of integration testing is to uncover the errors that might result because of the way units integrate or interface with each other.

Visualize the whole application as a hierarchy of components; integration testing can be performed in any of the following ways:

- **Bottom-up approach**—With this approach, the testing progresses from the smallest subsystem and then gradually progresses up in the hierarchy to cover the whole system. This approach may require you to write a number of test-driver programs that test the integration between subsystems.

- **Top-down approach**—This approach starts with the top-level system, to test the top-level interfaces, and gradually comes down and tests smaller subsystems. You might be required to write *stubs* (that is, dummy modules that mimic the interface of a module but have no functionality) for the modules that are not yet ready for testing.
- **Umbrella approach**—This approach focuses on testing the modules that have a high degree of user interaction. Normally, stubs are used in place of process-intensive modules. This approach enables you to release graphical user interface (GUI)-based applications early and allows you to gradually increase functionality. It is called the *umbrella approach* because when you look at the application hierarchy (as shown in Figure 12.1), the input/output modules are generally present on the edges, forming an umbrella shape.

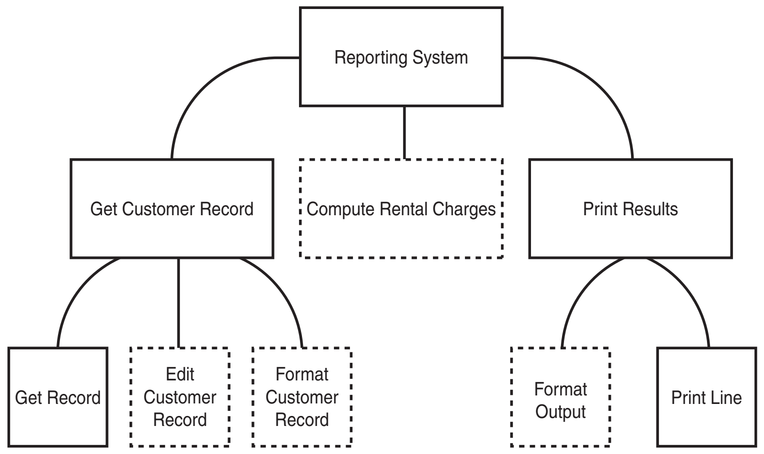


FIGURE 12.1
The umbrella approach of integration testing focuses on testing the modules that have a high degree of user interaction.

> NOTE
> **Limitations of Testing** Testing can show the presence of errors, but it can never confirm the absence of errors. Various factors such as the complexity of the software, requirements such as interoperability with various software and hardware, and globalization issues such as support for various languages and cultures, can create excessive input data and too many execution paths to be tested. Many companies do their best to capture most of the test cases by using automation (that is, using computer programs to find errors) and beta-testing (that is, involving product enthusiasts to find errors), but errors in final products are still a well-known and acknowledged fact.

Regression Testing

Regression testing should be performed any time a program is modified, either to fix a bug or to add a feature. The process of regression testing involves running all the tests mentioned in the preceding sections as well as any newly added test cases to test the added functionality. Regression testing has two main goals:

- Verify that all known bugs are corrected.
- Verify that the program has no new bugs.

Testing International Applications

Debug, rework, and resolve defects in code.

- **Provide multicultural test data to components and applications.**

Testing an application designed for international usage involves checking the country and language dependencies of each locale for which the application has been designed. When testing international applications, you need to consider the following:

- You should test the application's data and user interface to make sure that they conform to the locale's standards for date and time, numeric values, currency, list separators, and measurements for the countries in which you plan to sell your product.
- You should test your application on as many language and culture variants as necessary to cover your entire market for the application. Operating systems such as Windows 2000 and Windows XP support the languages used in more than 120 cultures/locales.
- You should use Unicode for your applications. Applications that use Unicode run without requiring any changes on Windows 2000 and XP. If an application instead uses Windows code pages, you need to set the culture/locale of the operating system according to the localized version of the application that you are testing. Each such change requires you to reboot the computer.
- While testing a localized version of an application, you should make sure to use the input data in the language that is supported by the localized version. This makes the testing scenario similar to the scenario in which the application will be actually used.

For more discussion about support for globalization in a Windows application, refer to Chapter 8, "Globalization."

REVIEW BREAK

- Testing is the process of executing a program with the intention of finding errors. You should design an effective test plan to ensure that your application is free from all likely defects and errors.
- Unit testing ensures that each unit of an application functions precisely as desired. It is the lowest level of testing.
- Integration testing ensures that different units of an application function as expected by the test plan after they are integrated.
- Whenever code is modified or a new feature is added in an application, you should run all the existing test cases, along with a new set of test cases, to check the new feature. This helps you develop robust applications.

TRACING

Debug, rework, and resolve defects in code.

- **Create and apply debugging code to components and applications.**

The process of testing can reveal the presence of errors in a program, but to find the actual cause of a problem, you sometimes need the program to generate information about its own execution. Analysis of this information may help you understand why the program is behaving in a particular way and may lead to possible resolution of the error.

This process of collecting information about program execution is called *tracing*. You trace a program's execution in Visual C# .NET by generating messages about the program's execution with the use of the `Debug` and `Trace` classes.

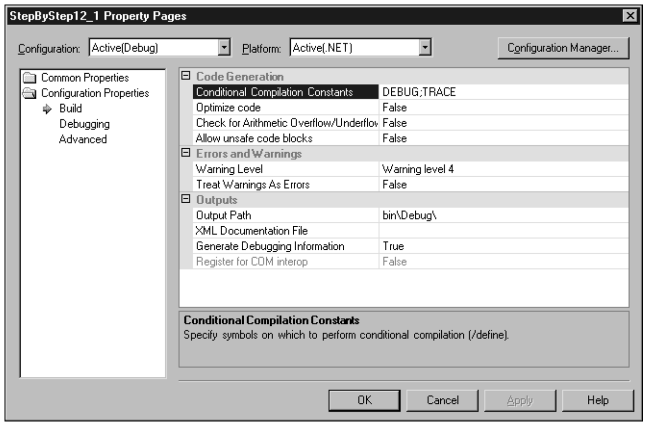
The `Trace` and `Debug` classes have several things in common:

- They both belong to the `System.Diagnostics` namespace.
- They have members with the same names.

- All their members are static.
- They are conditionally compiled (that is, their statements are included in the object code only if a certain symbol is defined).

The only difference between the `Debug` and `Trace` classes is that the members of the `Debug` class are conditionally compiled, but only when the `DEBUG` symbol is defined. On the other hand, members of the `Trace` class are conditionally compiled, but only when the `TRACE` symbol is defined.

Visual C# .NET provides two basic configurations for a project: `Debug` and `Release`. `Debug` is the default configuration. When you compile a program by using the `Debug` configuration, both `TRACE` and `DEBUG` symbols are defined, as shown in Figure 12.2. When you compile a program in the `Release` configuration, only the `TRACE` symbol is defined. You can switch between the `Debug` and `Release` configurations using the Solution Configurations combo box on the standard toolbar (as shown in Figure 12.3) or by using the Configuration Manager dialog box (as shown in Figure 12.4) from the project's Property Pages dialog box.



Configuration: Active(Debug)   Platform: Active(.NET)

Code Generation	
Conditional Compilation Constants	DEBUG;TRACE
Optimize code	False
Check for Arithmetic Overflow/Underflow	False
Allow unsafe code blocks	False
Errors and Warnings	
Warning Level	Warning level 4
Treat Warnings As Errors	False
Outputs	
Output Path	bin\Debug\
XML Documentation File	
Generate Debugging Information	True
Register for COM interop	False

Conditional Compilation Constants
Specify symbols on which to perform conditional compilation (/define).

FIGURE 12.2
Both the `TRACE` and `DEBUG` symbols are defined in the `Debug` configuration.

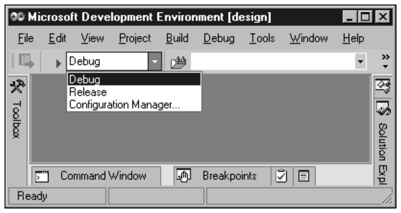


FIGURE 12.3
The standard toolbar of Visual Studio .NET contains a solutions configuration combo box to allow users to easily change solution configuration.

Later in this chapter, you will learn how to make these changes from within the program and through the command-line compilation options.

When you compile a program by using the `Debug` configuration, the code that uses the `Debug` and the `Trace` classes is included in the compiled code. When you run such a program, messages are generated by both the `Debug` and `Trace` classes. On the other hand, when a program is compiled by using the `Trace` configuration, it does not include any calls to the `Debug` class. Thus, when such a program is executed, you get only the messages generated by using the `Trace` class.

Table 12.1 summarizes the members of both the `Trace` and `Debug` classes.

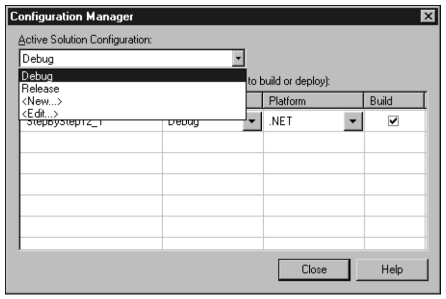


FIGURE 12.4
The Configuration Manager dialog box allows you to set configuration for projects in a solution.

NOTE

Tracing Helps in Resolving Hard-to-Reproduce Errors When programs run in a production environment, they sometimes report errors (mostly related to performance or threading problems) that are difficult to reproduce in a simulated testing environment. Tracing a production application can help you get runtime statistics for the program; this might help you in trapping these hard-to-reproduce errors.

TABLE 12.1

MEMBERS OF `Debug` AND `Trace` CLASSES

Member	*Type*	*Description*
`Assert()`	Method	Checks for a condition and displays a message if the condition is false.
`AutoFlush`	Property	Specifies whether the `Flush()` method should be called on the listeners after every write.
`Close()`	Method	Flushes the output buffer and then closes the listeners.
`Fail()`	Method	Displays an error message.
`Flush()`	Method	Flushes the output buffer and causes the buffered data to be written to the listeners.
`Indent()`	Method	Increases the current IndentLevel property by one.
`IndentLevel`	Property	Specifies the indent level.
`IndentSize`	Property	Specifies the number of spaces in an indent.
`Listeners`	Property	Specifies the collection of listeners that is monitoring the trace output.
`Unindent()`	Method	Decreases the current IndentLevel property by one.
`Write()`	Method	Writes the given information to the trace listeners in the Listeners collection.
`WriteIf()`	Method	Writes the given information to the trace listeners in the Listeners collection only if a condition is true.

continues

TABLE 12.1 *continued*

MEMBERS OF Debug AND Trace CLASSES

Member	*Type*	*Description*
`WriteLine()`	Method	Acts the same as Write(), but appends the information with a newline character.
`WriteLineIf()`	Method	Acts the same as `WriteIf()`, but appends the information with a newline character.

Using `Trace` and `Debug` to Display Information

Implement tracing.

- **Display trace output**

Step by Step 12.1 demonstrates how to use some of the methods of the `Trace` and `Debug` classes.

STEP BY STEP

12.1 Using the `Trace` and `Debug` Classes to Display Debugging Information

1. Launch Visual Studio .NET. Select File, New, Blank Solution and name the new solution `316C12`.

2. In Solution Explorer, right-click the name of solution and select Add, New Project. Select Visual C# Projects from the Project Types tree and then select Windows Application from the list of templates on the right. Name the project `StepByStep12_1`.

3. In Solution Explorer, right-click `Form1.cs` and rename it `FactorialCalculator`. Open the Properties window for this form and change its `Name` property to `FactorialCalculator` and Text property to `Factorial Calculator 12_1`. Switch to the code view of the form and modify the `Main()` method to launch `FactorialCalculator` instead of `Form1`.

4. Add two `TextBox` controls (`txtNumber` and `txtFactorial`) and a `Button` control (`btnCalculate`) to the form and arrange the controls as shown in Figure 12.5.

5. Add the following `using` directive in the code view:

```
using System.Diagnostics;
```

6. Add the following code to the `Click` event handler of `btnCalculate`:

```
private void btnCalculate_Click(object sender,
    System.EventArgs e)
{
    // write a debug message
    Debug.WriteLine(
       "Inside Button Click event handler");
    // start indenting messages now
    Debug.Indent();
    int intNumber = Convert.ToInt32(txtNumber.Text);
    // make a debug assertion
    Debug.Assert(intNumber >= 0, "Invalid value",
        "negative value in debug mode");
    // write a trace assertion
    Trace.Assert(intNumber >= 0, "Invalid value",
       "negative value in trace mode");

    int intFac = 1;
    for (int i = 2; i <= intNumber; i++)
    {
        intFac = intFac * i;
        // write a debug message
        Debug.WriteLine(i,
           "Factorial Program Debug, Value of i");
    }
    // write a trace message if the condition is true
    Trace.WriteLineIf(intFac < 1,
       "There was an overflow",
       "Factorial Program Trace");
    // write a debug message if the condition is true
    Debug.WriteLineIf(intFac < 1,
       "There was an overflow",
       "Factorial Program Debug");

    txtFactorial.Text = intFac.ToString();
    // decrease the indent level
    Debug.Unindent();

    // write a debug message
    Debug.WriteLine(
       "Done with computations, returning...");
}
```

continues

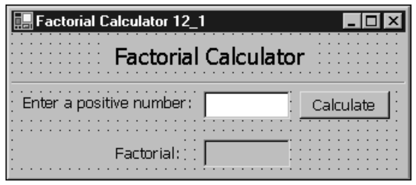
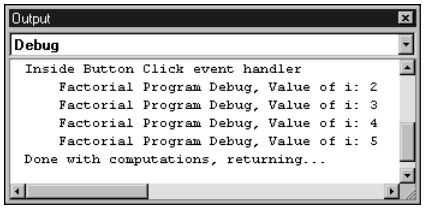


FIGURE 12.5
You can design a form that calculates the factorial of a given number.

continued

7. Run the project. Keep the program running and switch to the Visual Studio .NET Integrated Development Environment (IDE). Select View, Other Windows, Output. Push the pin on the title bar of the output window so that it does not hide automatically. Now switch to the running program; enter `5` in the text box and click the Calculate button. You should see `Debug` messages that are generated by the program (see Figure 12.6).

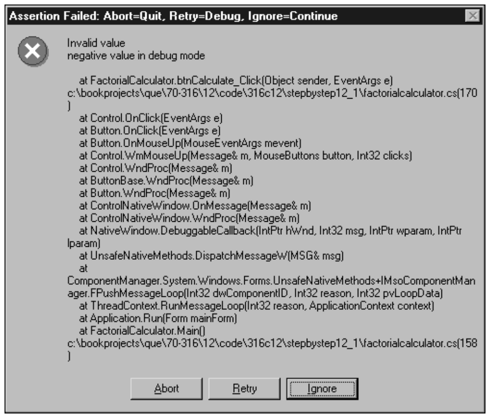


```
Inside Button Click event handler
    Factorial Program Debug, Value of i: 2
    Factorial Program Debug, Value of i: 3
    Factorial Program Debug, Value of i: 4
    Factorial Program Debug, Value of i: 5
Done with computations, returning...
```

FIGURE 12.6
`Debug` and `Trace` messages are by default always displayed in the Output window.

8. Now switch to the running program and enter the value `100` and click the Calculate button. Messages from both the `Debug` class and the `Trace` class overflow are displayed in the Output window. Note that the default configuration is the `Debug` configuration, where both the `TRACE` and `DEBUG` symbols are defined.

9. Enter a negative value, such as -1, and click the Calculate button. This causes the assertion to fail, and you see a dialog box that shows an assertion failed message, as shown in Figure 12.7. This message box is generated by the `Debug.Assert()` method in the code. The dialog box gives you three choices: Abort, to terminate the program; Retry, to break the program execution so that you can debug the program; and Ignore, to continue the execution as if nothing has happened. Click Ignore, and you see another Assertion Failed dialog box. This one was generated by the `Trace.Assert()` method in the code. Click the Abort button to terminate the program execution.



```
Invalid value
negative value in debug mode

    at FactorialCalculator.btnCalculate_Click(Object sender, EventArgs e)
c:\bookprojects\que\70-316\12\code\316c12\stepbystep12_1\factorialcalculator.cs(170
)
    at Control.OnClick(EventArgs e)
    at Button.OnClick(EventArgs e)
    at Button.OnMouseUp(MouseEventArgs mevent)
    at Control.WmMouseUp(Message& m, MouseButtons button, Int32 clicks)
    at Control.WndProc(Message& m)
    at ButtonBase.WndProc(Message& m)
    at Button.WndProc(Message& m)
    at ControlNativeWindow.OnMessage(Message& m)
    at ControlNativeWindow.WndProc(Message& m)
    at NativeWindow.DebuggableCallback(IntPtr hWnd, Int32 msg, IntPtr wparam, IntPtr
lparam)
    at UnsafeNativeMethods.DispatchMessageW(MSG& msg)
    at
ComponentManager.System.Windows.Forms.UnsafeNativeMethods+IMsoComponentMan
ager.FPushMessageLoop(Int32 dwComponentID, Int32 reason, Int32 pvLoopData)
    at ThreadContext.RunMessageLoop(Int32 reason, ApplicationContext context)
    at Application.Run(Form mainForm)
    at FactorialCalculator.Main()
c:\bookprojects\que\70-316\12\code\316c12\stepbystep12_1\factorialcalculator.cs(158
)
```

FIGURE 12.7
The Assertion Failed dialog box is displayed when an assertion that is made in the `Assert()` method fails.

10. From the Solution Configurations combo box on the standard toolbar, select the `Release` configuration. (The `Release` configuration defines only the `TRACE` symbol.) Run the program again. Enter the value `5` and click the Calculate button. The factorial is calculated, but no messages appear in the Output window. Enter the value `100` and click the Calculate button. You should now see the trace overflow message in the Output window. Finally, try calculating the factorial of `-1`. You should see just one dialog box, showing an assertion failed message. Click the Abort button to terminate the program.

Note from Step by Step 12.1 that you can use the methods of the `Debug` and `Trace` classes (for example, the `WriteIf()` and `WriteLineIf()` methods) to display messages based on conditions. This can be a very useful technique if you are trying to understand the flow of logic of a program. Step by Step 12.1 also demonstrates the use of the `Assert()` method. The `Assert()` method tests your assumption about a condition at a specific place in the program. When an assertion fails, the `Assert()` method pinpoints the code that is not behaving according to your assumptions. A related method is `Fail()`. The `Fail()` method displays a dialog box similar to the one that `Assert()` shows, but it does not work conditionally. `Fail()` signals unconditional failure in a branch of code execution.

Trace Listeners

Implement tracing.

- **Add trace listeners and trace switches to an application.**

Listeners are the classes that are responsible for forwarding, recording, and displaying the messages generated by the `Trace` and `Debug` classes. You can have multiple listeners associated with the `Trace` and `Debug` classes, by adding `Listener` objects to their `Listeners` property. The `Listeners` property is a collection that is capable of holding any objects derived from the `TraceListener` class. The `Debug` and `Trace` classes share a `Listeners` collection, so an object that is added to the `Listeners` collection of the `Debug` class is automatically available in the `Trace` class and vice versa.

The `TraceListener` class is an abstract class that belongs to the `System.Diagnostics` namespace and has three implementations:

- **`DefaultTraceListener`**—An object of this class is automatically added to the `Listeners` collection. Its behavior is to write messages on the Output window.
- **`TextWriterTraceListener`**—An object of this class writes messages to any class that derives from the `Stream` class and that includes the console or a file.
- **`EventLogTraceListener`**—An object of this class writes messages to the Windows event log.

EXAM TIP

The Same Listeners for Debug and Trace Messages sent through the Debug and Trace objects are directed through each Listener object in the Listeners collection. Debug and Trace share the same Listeners collection, so any Listener object that is added to the Trace.Listeners collection is also added to the Debug.Listeners collection.

If you want a listener object to perform differently from these three listener classes, you can create your own class that inherits from the TraceListener class. When doing so, you must at least implement the Write() and WriteLine() methods.

Step by Step 12.2 shows how to create a custom listener class that implements the TraceListener class to send debug and trace messages through email.

STEP BY STEP

12.2 Creating a Custom TraceListener Object

1. Create a new Windows application project in solution 316C12. Name the project StepByStep12_2.

2. In Solution Explorer, copy the FactorialCalculator.cs form from the StepByStep12_1 project to the current project. Change the Text property of the form to Factorial Calculator 12_2. Switch to the code view and change the namespace of the form to StepByStep12_2.

3. In Solution Explorer, right-click Form1.cs and select Delete from the context menu.

4. Add to the project a reference to System.Web.dll.

5. Using the Add Class Wizard, add a new class to the project. Name the class EmailTraceListener and add the following code to it (changing the From address to a valid email address):

```
using System;
using System.Diagnostics;
using System.Text;
using System.Web.Mail;

namespace StepByStep12_2
{
    public class EmailTraceListener : TraceListener
    {
        // Message log will be sent to this address
        private string mailto;
        // Store the message log
        private StringBuilder message;

        public EmailTraceListener(string mailto)
        {
            this.mailto = mailto;
        }
        // A custom listener must
        // override Write() method
        public override void Write(string message)
        {
            if (this.message == null)
                this.message = new StringBuilder();
            this.message.Append(message);
        }

        // A custom listener must
        // override WriteLine() method
        public override void WriteLine(string message)
        {
            if (this.message == null)
                this.message = new StringBuilder();
            this.message.Append(message);
            this.message.Append('\n');
        }

        // use the close method to send mail.
        public override void Close()
        {
            // ensure that the listener is flushed
            Flush();
            // MailMessage belongs to the
            // System.Web.Mail namespace
            // but can be used from
            // any managed application
            if (this.message != null)
            {
                // Create a MailMessage object
                MailMessage mailMessage =
                    new MailMessage();
                mailMessage.From =
                    "tracelistener@youraddress.com";
                mailMessage.To = this.mailto;
                mailMessage.Subject =
               "Factorial Program Debug/Trace output";
                mailMessage.Body =
                    this.message.ToString();
                //send the mail
                SmtpMail.Send(mailMessage);
            }
        }

        public override void Flush()
        {
            // nothing much to do here
            // so call the base class's implementation
            base.Flush();
        }
    }
}
```

continues

NOTE

Sending Email Messages The types in the `System.Web.Mail` namespace can be used from any managed application, including both Web and Windows applications. This functionality is supported only in the Windows 2000, Windows XP Professional, and Windows .NET Server operating systems. For other operating systems, you can send email messages by manually establishing Simple Mail Transfer Protocol (SMTP) connections through the `System.Net.TcpClient` class or by using an SMTP component, which you might be able to get from a component vendor for free.

continued

6. Add the following code to the `Load` event of the `FactorialCalculator` form, changing the email address *`Insert@youraddress.here`* to a real address that can receive emails:

```
private void FactorialCalculator_Load(object sender,
    System.EventArgs e)
{
    // Add a custom listener to
    // the Listeners collection
    Trace.Listeners.Add(new EmailTraceListener(
       "Insert@youraddress.here"));
}
```

7. Add the following code to the `Closing` event of the `FactorialCalculator` form:

```
private void FactorialCalculator_Closing(
     object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // call the Close() method for all listeners
    Trace.Close();
}
```

8. Set project `StepByStep12_2` as the startup project.

9. Run the project, using the default `Debug` configuration. Enter a value and click the Calculate button. Close the form. Note that both `Debug` and `Trace` messages appear on the Output window, and they are emailed to the specified address by using the local SMTP server. Run the project again in the `Release` mode. Enter a large value, such as `100`, and click the Calculate button. The overflow message appears in the Output window. Close the form. While you are closing the form, an email message containing the `Trace` messages are sent to the specified email address.

Trace Switches

Implement tracing.

- **Add trace listeners and trace switches to an application.**

So far in this chapter, you have learned that the `Trace` and `Debug` classes can be used to display valuable information related to program execution. You have also learned that it is possible to capture messages in a variety of formats. In this section, you will learn how to control the nature of messages that you want to get from a program.

You can use trace switches to set the parameters that control the level of tracing that needs to be done on a program. You set these switches via an Extensible Markup Language (XML) based external configuration file. This is especially useful when the application you are working with is in production mode. You might initially want the application not to generate any trace messages. However, if the application later has problems or you just want to check on the health of the application, you might want to instruct the application to emit a particular type of trace information by just changing the configuration file. You are not required to recompile the application; the application automatically picks up the changes from the configuration file when it restarts.

There are two predefined classes for creating trace switches: the `BooleanSwitch` class and the `TraceSwitch` class. Both of these classes derive from the abstract `Switch` class. You can also define your own trace switch classes by deriving classes from the `Switch` class.

You use the `BooleanSwitch` class to differentiate between two modes of tracing: trace-on and trace-off. Its default value is zero, which corresponds to the trace-off state. If it is set to any nonzero value, it corresponds to the trace-on state.

Unlike `BooleanSwitch`, the `TraceSwitch` class provides five different levels of tracing switches. These levels are defined by the `TraceLevel` enumeration and are listed in Table 12.2. The default value of `TraceLevel` for a `TraceSwitch` object is `0 (Off)`.

EXAM TIP

Out-of-Range Values for `BooleanSwitch` and `TraceSwitch` For a `BooleanSwitch` object, if any nonzero (negative or positive) value is specified in the configuration file, the `BooleanSwitch` object's `Enabled` property is set to `true`. For a `TraceSwitch` object, if a value greater than 4 is specified, the `Level` property of the object is set to `TraceLevel.Verbose (4)`. If a negative value is specified, a `StackOverflow` exception occurs at runtime.

TABLE 12.2

THE `TraceLevel` ENUMERATION

Enumerated Value	*Integer Value*	*Type of Tracing*
`Off`	0	None
`Error`	1	Only error messages
`Warning`	2	Warning messages and error messages
`Info`	3	Informational messages, warning messages, and error messages
`Verbose`	4	Verbose messages, informational messages, warning messages, and error messages

Table 12.3 displays the important properties of the `TraceSwitch` class.

TABLE 12.3

IMPORTANT PROPERTIES OF THE `TraceSwitch` CLASS

Property	*Description*
`Description`	Describes the switch (inherited from `Switch`).
`DisplayName`	Specifies a name used to identify the switch (inherited from `Switch`).
`Level`	Specifies the trace level that helps select which trace and debug messages will be processed. Its value is one of the TraceLevel enumeration values (refer to Table 12.2).
`TraceError`	Returns `true` if `Level` is set to `Error`, `Warning`, `Info`, or `Verbose`; otherwise, it returns `false`.
`TraceInfo`	Returns `true` if `Level` is set to `Info` or `Verbose`; otherwise, it returns `false`.
`TraceVerbose`	Returns `true` if `Level` is set to `Verbose`; otherwise, it returns `false`.
`TraceWarning`	Returns `true` if `Level` is set to `Warning`, `Info`, or `Verbose`; otherwise, it returns `false`.

Step by Step 12.3 demonstrates how to use trace switches in a Windows application.

STEP BY STEP

12.3 Using the `TraceSwitch` Class

1. Create a new Windows application project in solution `316C12`. Name the project `StepByStep12_3`.
2. In Solution Explorer, copy the `FactorialCalculator.cs` form from the `StepByStep12_1` project to the current project. Change the `Text` property of the form to `Factorial Calculator 12_3`. Switch to the code view and change the namespace of the form to `StepByStep12_3`.
3. Delete the default `Form1.cs`.
4. Declare the following static variable at the class level, just after the `Main()` method:

```
static TraceSwitch traceSwitch =
       new TraceSwitch("FactorialTrace",
       "Trace the factorial application");
```

5. Change the `Click` event handler of the Calculate button so that it has the following code:

```
private void btnCalculate_Click(object sender,
   System.EventArgs e)
{
    if (traceSwitch.TraceVerbose)
        // write a debug message
        Debug.WriteLine(
           "Inside the Button Click event handler");

    // start indenting messages now
    Debug.Indent();
    int intNumber = Convert.ToInt32(txtNumber.Text);

    if (traceSwitch.TraceError)
    {
        // make a debug assertion
        Debug.Assert(intNumber >= 0, "Invalid value",
           "negative value in debug mode");
    }

    int intFac = 1;
    for (int i = 2; i <= intNumber; i++)
    {
        intFac = intFac * i;
        // write a debug message
        if (traceSwitch.TraceInfo)
            Debug.WriteLine(i,
               "Factorial Program Debug, Value of i");
    }
```

continues

continued

```
    if (traceSwitch.TraceWarning)
        // write a debug message
        // if the condition is true
        Debug.WriteLineIf(intFac < 1,
            "There was an overflow",
            "Factorial Program Debug");

    txtFactorial.Text = intFac.ToString();
    // decrease the indent level
    Debug.Unindent();

    if (traceSwitch.TraceVerbose)
        // write a debug message
        Debug.WriteLine(
           "Done with computations, returning...");
}
```

6. In Solution Explorer, select View All Files from the toolbar. Navigate to the `bin\debug` folder. Right-click the `debug` folder and then select Add, Add New Item. Choose to create an XML file and name the XML file `StepByStep12_3.exe.config`.

7. In the XML editor, type the following configuration data in the XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.diagnostics>
        <switches>
            <add name="FactorialTrace" value="4" />
        </switches>
    </system.diagnostics>
</configuration>
```

8. Set project `StepByStep12_3` as the startup project.

9. Run the project, using the default `Debug` configuration. Enter the value `5`; note that all messages appear in the output window. Enter a negative value and then a large value, and you see all the errors and warning messages. Close the form. Modify the XML file to change the value of `FactorialTrace` to `3`. Run the project again, you should now see all messages except the one set with `TraceLevel` as `Verbose`. Repeat the process, with values of `FactorialTrace` in the configuration file changed to `2`, `1`, and `0`.

10. Modify the program to change all `Debug` statements to `Trace` statements. Copy the XML configuration file to the `bin\Release` folder in the project and then repeat step 9, using the `Release` configuration.

Conditional Compilation

The C# programming language provides a set of preprocessing directives. You can use these directives to skip sections of source files for compilation, to report errors and warnings, or to mark distinct regions of source code.

Table 12.4 summarizes the preprocessing directives that are available in C#.

NOTE

C# and the Preprocessor There is no separate preprocessor in the Visual C# .NET compiler. The lexical analysis phase of the compiler processes all the preprocessing directives. C# uses the term *preprocessor* from a conventional point of view, in contrast to languages such as C and C++ that have separate preprocessors for taking care of conditional compilation.

TABLE 12.4

C# PREPROCESSING DIRECTIVES

Directives	*Description*
`#if`, `#else`, `#elif`, and `#endif`	These directives conditionally skip the sections of code. The skipped sections are not part of the compiled code.
`#define` and `#undef`	These directives define or undefine symbols in the code.
`#warning` and `#error`	These directives explicitly generate error or warning messages. The compiler reports errors and warnings in the same way it reports other compile-time errors and warnings.
`#line`	This directive alters the line numbers and source file filenames reported by the compiler in warning and error messages.
`#region` and `#endregion`	These directives mark sections of code. A common example of these directives is the code generated by Windows Forms Designer. Visual designers such as Visual Studio .NET can use these directives to show, hide, and format code.

In addition to providing preprocessing directives, the C# programming language also provides a `ConditionalAttribute` class.

You can mark a method as conditional by applying the `Conditional` attribute to it. The `Conditional` attribute takes one argument that specifies a symbol. The conditional method is either included or omitted from the compiled code, depending on the definition of the specified symbol at that point. If the symbol definition is available, the code of the method is included; otherwise, the code of the method is excluded from the compiled code.

EXAM TIP

Conditional Methods A method must have its return type set to `void` in order to have the `Conditional` attribute applied to it.

The conditional compilation directives and methods with the `Conditional` attribute allow you to keep debugging-related code in the source code but exclude it from the compiled version. This removes the extraneous messages and the production programs do not encounter performance hits due to processing of additional code. In this case, if you want to resolve some errors, you can easily activate the debugging code by defining a symbol and recompiling the program.

Step by Step 12.4 demonstrate the use of `ConditionalAttribute` and the conditional compilation directives.

STEP BY STEP

12.4 Using Conditional Compilation

1. Create a new Windows application project in solution `316C12`. Name the project `StepByStep12_4`.
2. In Solution Explorer, copy the `FactorialCalculator.cs` form from the `StepByStep12_1` project to the current project. Set the `Text` property of the form to `Factorial Calculator 12_4`. Switch to the code view and change the namespace of the form to `StepByStep12_4`.
3. Delete the default `Form1.cs`.
4. Add the following two conditional methods to the class definition:

```
[Conditional("DEBUG")]
public void InitializeDebugMode()
{
    label1.Text = "Factorial Calculator: Debug Mode";
}
```

```
 [Conditional("TRACE")]
public void InitializeReleaseMode()
{
    label1.Text = "Factorial Calculator Version 1.0";
}
```

5. Attach an event handler to the form's `Load` event and add the following code:

```
private void StepByStep12_4_Load(
    object sender, System.EventArgs e)
{
    #if !DEBUG && !TRACE
       #error you should have either
➥DEBUG or TRACE defined
    #endif

    #if DEBUG
        Debug.WriteLine(
            "Program started in debug mode");
        InitializeDebugMode();
    #else
        Trace.WriteLine(
            "Program started in release mode");
        InitializeReleaseMode();
    #endif
}
```

6. Set project `StepByStep12_4` as the startup project.

7. Run the project, using the default `Debug` configuration. The heading of the form displays "Factorial Program: Debug Mode" (see Figure 12.8). The Output window also displays a string: "Program started in debug mode." Close the program and start it again in the `Release` mode. A different heading appears in the form (see Figure 12.9) and a different message appears in the Output window.

8. Add the following line as the very first line of the code:

```
#undef DEBUG
```

Run the program, using the `Debug` configuration. Note that the program is executed as if it were executed in the `Trace` configuration. This is because the `Debug` configuration defines both the `DEBUG` and `TRACE` symbols. Because `DEBUG` is undefined using the `#undef` preprocessing directive in the code you added, the compiled code includes the `#else` part of the preprocessing directive.

continues

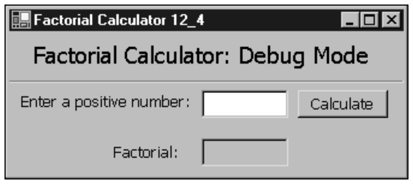


FIGURE 12.8
The factorial calculator can be conditionally compiled by using the `Debug` configuration.

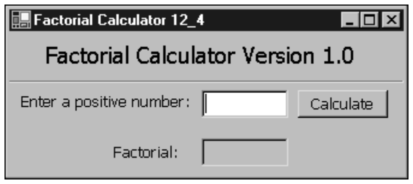


FIGURE 12.9
The factorial calculator can be conditionally compiled by using the `Release` configuration.

continued

9. Add the following line just after the directive placed in step 8:

```
#undef TRACE
```

   Try running the program. Rather than run the program, the compiler throws an error message, complaining that both `DEBUG` and `TRACE` are undefined. This message is caused by the conditional logic in the `Load` event handler of the form.

You can define the `DEBUG` and `TRACE` symbols for the compiler in the following ways:

- By defining the constants in the project's property pages dialog box
- By using the `#define` directive at the beginning of the code file
- By using the `/define` (`/d` for short) option with the command-line C# compiler

Step by Step 12.4 demonstrates conditional compilation with the `DEBUG` and `TRACE` symbols. You can also use conditional compilation with any other custom-defined symbols to perform conditional compilation.

GUIDED PRACTICE EXERCISE 12.1

The goal of this exercise is to add an `EventLogTraceListener` object to the Factorial Calculator program so that it will write all `Trace` and `Debug` messages to the Windows event log.

This exercise will give you good practice using trace listeners. How would you create such a form?

You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Create a new Visual C# Windows application in solution `316C12`. Name the project `GuidedPracticeExercise12_1`.

2. In Solution Explorer, copy the `FactorialCalculator.cs` form from the `StepByStep12_1` project to the current project. Set the `Text` property of the form to `Factorial Calculator GuidedPracticeExercise12_1`. Switch to the code view and change the namespace of the form to `GuidedPracticeExercise12_1`.

3. Delete the default `Form1.cs`.

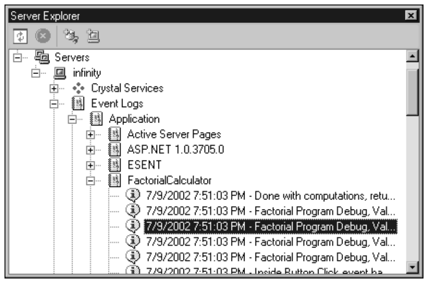
4. Double-click the form to add an event handler for the `Load` event. Add the following code to the event handler:

```
private void FactorialCalculator_Load(object sender,
    System.EventArgs e)
{
    //Add a event log listener to
    //the Listeners collection
    Trace.Listeners.Add(new EventLogTraceListener(
       "FactorialCalculator"));
}
```

5. Set project `GuidedPracticeExercise12_1` as the startup project.

6. Run the project. Enter a value for finding a factorial. Click the Calculate button. Close the program. Select View, Server Explorer. Navigate to your computer, and expand the `Event Logs` node, the `Application` node, and the `FactorialCalculator` node. The messages generated by the `Trace` and `Debug` classes are added to the Application event log, as shown in Figure 12.10.

If you have difficulty following this exercise, review the sections "Trace Listeners" and "Using Trace and Debug to Display Information," earlier in this chapter. After doing that review, try this exercise again.



```
7/9/2002 7:51:03 PM - Done with computations, retu...
7/9/2002 7:51:03 PM - Factorial Program Debug, Val...
7/9/2002 7:51:03 PM - Factorial Program Debug, Val...
7/9/2002 7:51:03 PM - Factorial Program Debug, Val...
7/9/2002 7:51:03 PM - Factorial Program Debug, Val...
7/9/2002 7:51:03 PM - Inside Button Click event ha...
```

FIGURE 12.10
You can view the Windows Event Log from Server Explorer.

R E V I E W   B R E A K

- The `Trace` and `Debug` classes can be used to display informative messages in an application when the `DEBUG` and `TRACE` symbols are defined, respectively, at the time of compilation.
- By default, both `TRACE` and `DEBUG` symbols are defined in the `Debug` configuration for compilation, and only the `TRACE` symbol is defined for the `Release` configuration for compilation.
- Listeners are objects that receive trace and debug output. By default, both `Trace` and `Debug` classes have the `DefaultTraceListener` object in their `Listeners` collections. The `DefaultTraceListener` object displays messages in the Output window.
- `Debug` and `Trace` objects share the same `Listeners` collection. Therefore, any listener object added to the `Trace.Listeners` collection is also added to the `Debug.Listeners` collection.
- Trace switches allow you to change the type of messages traced by a program, depending on the value stored in the XML configuration file. You need not recompile the application for this change to take effect; you just restart it. You need to implement code to display the messages, depending on the value of the switch.
- C# preprocessing directives allow you to define and undefine symbols in an application, report errors or warnings, mark regions of code, and conditionally skip code for compilation.
- The `Conditional` attribute allows you to conditionally add or skip a method for compilation, depending on the value of the symbol that is passed as a parameter to the attribute.

DEBUGGING

Debug, rework, and resolve defects in code.

- **Configure the debugging environment.**

Debugging is the process of finding the causes of errors in a program, locating the lines of code that are causing those errors, and fixing those errors.

Without tools, the process of debugging can be very time-consuming and tedious. Thankfully, Visual Studio .NET comes loaded with a large set of tools to help you with various debugging tasks.

NOTE

Runtime Errors and Compile-Time Errors *Compile-time errors* are produced when a program does not comply with the syntax of the programming language. These errors are trivial and are generally pointed out by compilers themselves. *Runtime errors* occur in programs that are compiled successfully but do not behave as expected. The process of testing and debugging applies to runtime errors only. Testing reveals these errors, and debugging repairs them.

Stepping Through Program Execution

A common technique for debugging is to execute a program step-by-step. This systematic execution allows you to track the flow of logic, to ensure that the program is following the same paths of execution that you expect it to follow. If it does not, you can immediately identify the location of the problem.

Using step-by-step execution of a program also gives you an opportunity to monitor the program's state before and after a statement is executed. For example, you can check the values of variables, the records in a database, and other changes in the environment. Visual Studio .NET provides tools to make these tasks convenient.

The Debug menu provides three options for step-by-step execution of a program (see Table 12.5). The keyboard shortcuts listed in Table 12.5 correspond to the default keyboard scheme of the Visual Studio .NET IDE. If you have personalized the keyboard scheme either through the Tools, Options, Environment, Keyboard menu or through the Visual Studio . NET Start Page, you might have a different keyboard mapping. You can check out the keyboard mappings available for your customization through Visual Studio .NET's context-sensitive help.

TABLE 12.5

DEBUG OPTIONS FOR STEP-BY-STEP EXECUTION

Debug Menu Item	*Keyboard Shortcut*	*Purpose*
Step Into	F11	You use this option to execute code in step mode. If a method call is encountered, the program execution steps into the code of the method and executes the method in step mode.

continues

TABLE 12.5 *continued*

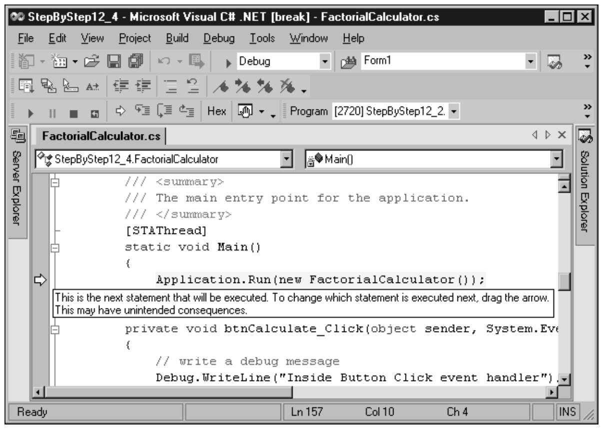
DEBUG OPTIONS FOR STEP-BY-STEP EXECUTION

Debug Menu Item	*Keyboard Shortcut*	*Purpose*
Step Over	F10	You use this option when a method call is encountered and you do not want to step into the method code. When this option is selected, the debugger executes the entire method without any step-by-step execution (interruption), and then it steps to the next statement after the method call.
Step Out	Shift+F11	You use this option inside a method call to execute the rest of the method without stepping, and you resume step execution mode when control returns to the calling method.

STEP BY STEP

12.5 Trying Step-by-Step Execution of a Windows Application

1. Set project `StepByStep12_4` as the startup project.
2. Select Debug, Step Into. The program pauses its execution at the first executable statement and shows the statement highlighted, as shown in Figure 12.11. An arrow appears in the left margin of the code, and it points at the next statement to be executed.
3. Press F11 to proceed to the next step. The debugger steps into the Windows Form Designer Generated Code section, where it executes the constructor code of the `FactorialCalculator` form to create its new instance, as requested by the `Application.Run()` method. Press F11 a couple times.


StepByStep12_4 - Microsoft Visual C# .NET [break] - FactorialCalculator.cs
File Edit View Project Build Debug Tools Window Help
Debug
Form1
Hex
Program [2720] StepByStep12_2.
FactorialCalculator.cs
StepByStep12_4.FactorialCalculator
Main()
Server Explorer
Solution Explorer
Ready
Ln 157 Col 10 Ch 4 INS

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new FactorialCalculator());
```

This is the next statement that will be executed. To change which statement is executed next, drag the arrow. This may have unintended consequences.

```
        private void btnCalculate_Click(object sender, System.Eve
        {
            // write a debug message
            Debug.WriteLine("Inside Button Click event handler").
```

FIGURE 12.11
You can step through a program's execution by selecting Debug, StepInto from the Visual Studio .NET menu.

4. Drag the yellow arrow back one line. In this way you can instruct the debugger to change what statement will be executed next. Press F11 two times to see the effect of dragging the debugger back. Now press Shift+F11. This automates the execution for the rest of the current method, and the execution breaks again at the next statement to be executed in the calling code.

5. Control comes back to the `Application.Run()` method call. The form is now created and is ready to be launched, as soon as you initiate the next step by pressing F11. Press the F11 key, and you see the form execute.

6. Enter a positive number in the form and press the Calculate button. The form calculates the factorial and displays it almost instantly. Note that the application is no longer running in step mode. Pressing F11 either on the form or in the code view has no effect.

The lesson from Step by Step 12.5 is that when you start an application in step mode, after the `Application.Run()` method is executed and the form is launched, you cannot really go back to step-by-step execution of the code. To step into the code of various event handlers of a form, you need to mark breakpoints in the code, as described in the following section.

Setting Breakpoints

Breakpoints are markers in code that signal the debugger to pause execution. When the debugger pauses at a breakpoint, you can take your time to analyze variables, data records, and other settings in the environment to determine the state of the program. You can also choose to execute the program in step mode from this point onward.

If you have placed a breakpoint in the `Click` event handler of a button, the program pauses when you click the button and the execution reaches the breakpoint. You can then step through the execution for the rest of the event handler. When the execution of the event handler code finishes, the control is transferred back to the form. If you have another button on the form for which a breakpoint is not set in the event handler, then the program is no longer under step execution. You should mark breakpoints at all the places you would like execution to pause.

STEP BY STEP

12.6 Working with Breakpoints

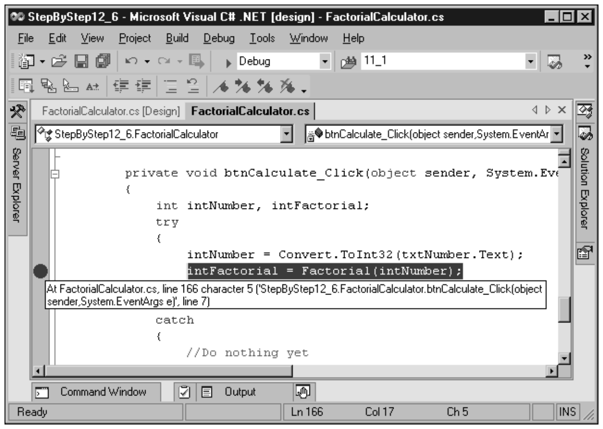
1. Create a new Windows application project in solution `316C12`. Name the project `StepByStep12_6`.
2. In Solution Explorer, copy the `FactorialCalculator.cs` form from the `StepByStep12_1` project to the current project. Change the `Text` property of the form to `Factorial Calculator 12_6`. Switch to the code view and change the namespace of the form to `StepByStep12_6`.
3. Delete the default `Form1.cs`.
4. Set project `StepByStep12_6` as the startup project.
5. Add the following method to the class:

```
private int Factorial(int intNumber)
{
    int intFac = 1;
    for (int i = 2; i <= intNumber; i++)
    {
        intFac = intFac * i;
    }
    return intFac;
}
```

6. Modify the `Click` event handler of `btnCalculate` so that it looks like this:

```
private void btnCalculate_Click(object sender,
   System.EventArgs e)
{
    int intNumber, intFactorial;
    try
    {
        intNumber = Convert.ToInt32(txtNumber.Text);
        intFactorial = Factorial(intNumber);
        txtFactorial.Text = intFactorial.ToString();
    }
    catch(Exception ex)
    {
        Debug.WriteLine(ex.Message);
    }
}
```

7. In the event handler added in step 6, right-click the beginning of the line that makes a call to the `Factorial()` method and select Insert Breakpoint from the context menu. Note that the line of code is highlighted with red and that a red dot appears in the left margin, as in Figure 12.12. Alternatively, you could create a breakpoint by clicking the left margin adjacent to a line.


StepByStep12_6 - Microsoft Visual C# .NET [design] - FactorialCalculator.cs
File Edit View Project Build Debug Tools Window Help
Debug
11_1
FactorialCalculator.cs [Design]
FactorialCalculator.cs
StepByStep12_6.FactorialCalculator
btnCalculate_Click(object sender,System.EventAr
Server Explorer
Solution Explorer
Command Window
Output
Ready
Ln 166
Col 17
Ch 5
INS

```
        private void btnCalculate_Click(object sender, System.Eve
        {
            int intNumber, intFactorial;
            try
            {
                intNumber = Convert.ToInt32(txtNumber.Text);
                intFactorial = Factorial(intNumber);
At FactorialCalculator.cs, line 166 character 5 ('StepByStep12_6.FactorialCalculator.btnCalculate_Click(object
sender,System.EventArgs e)', line 7)
            catch
            {
                //Do nothing yet
```

FIGURE 12.12
You can enter step-by-step execution mode by setting a breakpoint in a program.

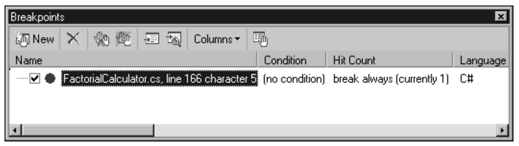
NOTE

The Disassembly Window Shows Native Code Instead of MSIL
Although C# programs are compiled to Microsoft Intermediate Language (MSIL), they are just-in-time compiled to native code only at the time of their first execution. This means the executing code is never in IL; it is always in native code. Thus, you will always see native code instead of IL in the Disassembly window.

8. Execute the project. The Factorial form appears. Enter a value and click the Calculate button. Note that execution pauses at the location where you have marked the breakpoint.

continues

9. Press F11 to step into the code of the `Factorial()` method. Move the mouse pointer over various variables in the `Factorial()` method, and you see the current values of these variables.
10. Select Debug, Windows, Breakpoints. The Breakpoints window appears, as shown in Figure 12.13. Right-click the breakpoint listed in the window and select Goto Disassembly. The Disassembly window appears, showing the object code of the program along with the disassembled source code.



Name	Condition	Hit Count	Language
✓ ● FactorialCalculator.cs, line 166 character 5	(no condition)	break always (currently 1)	C#

FIGURE 12.13
The Breakpoints window gives you convenient access to all breakpoint-related tasks in one place.

11. Close the Disassembly window. Select Debug, Step Out to automatically execute the rest of the `Factorial()` method and again start the step mode in the event handler at the next statement. Step through the execution until you see the form again.
12. Select Debug, Stop Debugging. The debugging session ends and the application is terminated.
13. In the code view, right-click the statement where you have set the breakpoint and select Disable Breakpoint from the context menu.

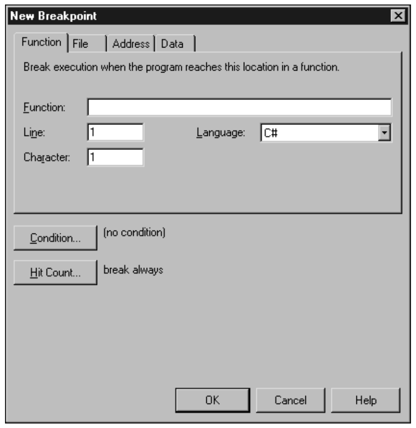
NOTE

The `Debug` Configuration Breakpoints and other debugging features are available only when you compile a program by using the `Debug` configuration.

NOTE

Disabling Versus Removing a Breakpoint When you remove a breakpoint, you loose all the information related to it. Instead of removing a breakpoint, you can choose to disable it. Disabling a breakpoint does not pause the program at the point of the breakpoint, but Visual C# .NET will remember the breakpoint settings. At any time, you can select Enable Breakpoint to reactivate the breakpoint.

To set advanced options in a breakpoint, you can choose to create a new breakpoint by selecting New from the context menu of the code or from the toolbar in the Breakpoints window. The New Breakpoint dialog box (see Figure 12.14) has four tabs. You can use these tabs to set a breakpoint in a function, in a file, at an address in the object code, and when a data value (that is, the value of a variable) changes.

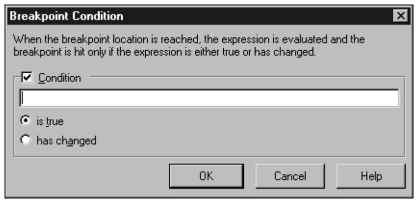


Break execution when the program reaches this location in a function.

Function:
Line: 1
Language: C#
Character: 1
Condition... (no condition)
Hit Count... break always

FIGURE 12.14◀
The New Breakpoint dialog box allows you to create a new breakpoint.

Clicking the Condition button opens the Breakpoint Condition dialog box, as shown in Figure 12.15. The Breakpoint Condition dialog box allows you to set a breakpoint based on the runtime value of an expression.

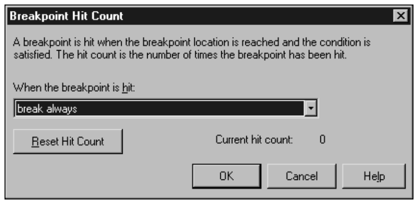


When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is either true or has changed.

☑ Condition
[✓ is true]
has changed

FIGURE 12.15▲
The Breakpoint Condition dialog box allows you to set a breakpoint that is based on the value of an expression at runtime.

Clicking the Hit Count button opens the Breakpoint Hit Count dialog box, as shown in Figure 12.16. This dialog box enables you to break the program execution only if the specified breakpoint has been hit a given number of times. This can be especially helpful if you have a breakpoint inside a lengthy loop and you want to step-execute the program only near the end of the loop.



A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit: break always
Current hit count: 0

FIGURE 12.16▲
The Breakpoint Hit Count dialog box enables you to break program execution only if the specified breakpoint has been hit a given number of times.

Analyzing Program State to Resolve Errors

Debug, rework, and resolve defects in code.

- **Resolve errors and rework code.**

When you break the execution of a program, the program is at a particular state in its execution cycle. You can use various debugging tools to analyze the values of variables, the results of expressions, the path of execution, and so on, to help identify the cause of the error that you are debugging.

Step by Step 12.7 demonstrates various Visual C# .NET debugging tools, such as the Watch, Autos, Locals, This, Immediate, Output and the Call Stack windows.

STEP BY STEP

12.7 Analyzing Program State to Resolve Errors

1. Create a new Windows application project in solution `316C12`. Name the project `StepByStep12_7`.

2. In Solution Explorer, copy the `FactorialCalculator.cs` form from the `StepByStep12_6` project to the current project. Change the `Text` property of the form to `Factorial Calculator 12_7`. Switch to the code view and change the namespace of the form to `StepByStep12_7`.

3. Delete the default `Form1.cs`.

4. Set project `StepByStep12_7` as the startup project.

5. Change the code in the `Factorial()` method to the following:

```
private int Factorial(int intNumber)
{
    int intFac = 1;
    for (int i = 2; i < intNumber; i++)
    {
        intFac = intFac * i;
    }
    return intFac;
}
```

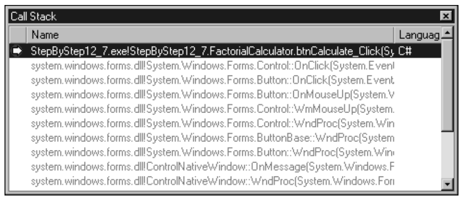
   Note in this code that I have introduced a logical error that I will later “discover” through debugging.

6. Run the program. Enter the value `5` in the text box and click the Calculate button. You should see that the result is not correct; this program needs to be debugged.

7. Set a breakpoint in the `Click` event handler of `btnCalculate` at the line where a call to the `Factorial()` method is being made. Execute the program. Enter the value `5` again, and click the Calculate button.

8. Press the F11 key to step into the `Factorial()` method. Select Debug, Windows, Watch, Watch1 to add a Watch window. Similarly, select the Debug, Windows menu and add the Locals, Autos, This, Immediate, Output and Call Stack windows. Pin down the windows so that they always remain in view and are easy to watch as you step through the program.

9. Look at the Call Stack window shown in Figure 12.17. It shows the method call stack, giving you information about the path taken by the code to reach its current point of execution. The currently executing method is at the top of the stack, as indicated by a yellow arrow. When this method is finished executing, the next entry in the stack will be the method receiving the control of execution.

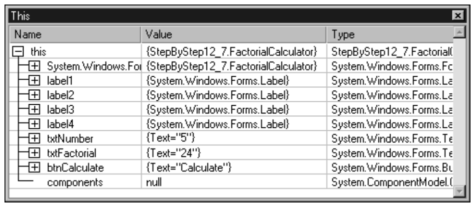


Call Stack

	Name	Languag
➡	StepByStep12_7.exe!StepByStep12_7.FactorialCalculator.btnCalculate_Click(Sy	C#
	system.windows.forms.dll!System.Windows.Forms.Control::OnClick(System.Even	
	system.windows.forms.dll!System.Windows.Forms.Button::OnClick(System.Event	
	system.windows.forms.dll!System.Windows.Forms.Button::OnMouseUp(System.V	
	system.windows.forms.dll!System.Windows.Forms.Control::WmMouseUp(System.	
	system.windows.forms.dll!System.Windows.Forms.Control::WndProc(System.Win	
	system.windows.forms.dll!System.Windows.Forms.ButtonBase::WndProc(System	
	system.windows.forms.dll!System.Windows.Forms.Button::WndProc(System.Win	
	system.windows.forms.dll!ControlNativeWindow::OnMessage(System.Windows.F	
	system.windows.forms.dll!ControlNativeWindow::WndProc(System.Windows.For	

FIGURE 12.17▲
The Call Stack window enables you to view the names of methods on the call stack, parameter types, and their values.

10. Look at the This window, shown in Figure 12.18. In the This window you can examine the members associated with the current object (the Factorial form). You can scroll down to find the `txtNumber` object. You can change the values of these objects here. At this point, you don't need to change any values.

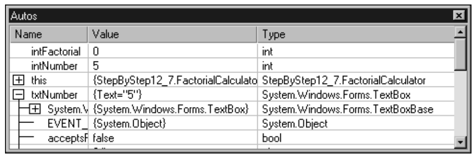


This

Name	Value	Type
⊟ this	{StepByStep12_7.FactorialCalculator}	StepByStep12_7.Factorial(
⊞ System.Windows.For	{StepByStep12_7.FactorialCalculator}	System.Windows.Forms.Fc
⊞ label1	{System.Windows.Forms.Label}	System.Windows.Forms.La
⊞ label2	{System.Windows.Forms.Label}	System.Windows.Forms.La
⊞ label3	{System.Windows.Forms.Label}	System.Windows.Forms.La
⊞ label4	{System.Windows.Forms.Label}	System.Windows.Forms.La
⊞ txtNumber	{Text="5"}	System.Windows.Forms.Te
⊞ txtFactorial	{Text="24"}	System.Windows.Forms.Te
⊞ btnCalculate	{Text="Calculate"}	System.Windows.Forms.Bu
components	null	System.ComponentModel.(

FIGURE 12.18▲
The This window enables you to examine the members associated with the current object.

11. Activate the Autos window, which is shown in Figure 12.19. The Autos window displays the variables used in the current statement and the previous statement. The debugger determines this information for you automatically; that is why the name of this window is Autos.

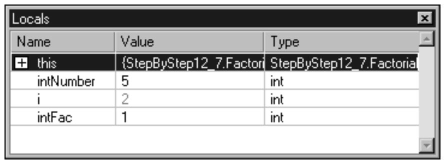


Autos

Name	Value	Type
intFactorial	0	int
intNumber	5	int
⊞ this	{StepByStep12_7.FactorialCalculato	StepByStep12_7.FactorialCalculator
⊟ txtNumber	{Text="5"}	System.Windows.Forms.TextBox
⊞ System.V	{System.Windows.Forms.TextBox}	System.Windows.Forms.TextBoxBase
EVENT_	{System.Object}	System.Object
acceptsF	false	bool

FIGURE 12.19◀
The Autos window displays the variables that are used in the current statement and the previous statement.

12. Invoke the Locals window, which is shown in Figure 12.20. The Locals window displays the variables that are local to the current context (that is, the current method under execution) with their current values. Figure 12.20 shows the local variables in the `Factorial()` method.



Locals

Name	Value	Type
⊞ this	{StepByStep12_7.Factori	StepByStep12_7.Factorial
intNumber	5	int
i	2	int
intFac	1	int

FIGURE 12.20▲
The Locals window displays the variables that are local to the method currently under execution.

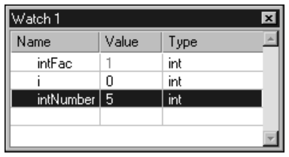
13. Invoke the Immediate window. Type `intNumber` in the Immediate window and press Enter. The Immediate window immediately evaluates and displays the current value of this variable in the next line. Now type the expression `Factorial(intNumber)`. The Immediate window calls the `Factorial()` method for a given value and prints the result.

continues

continued

The Immediate window can therefore be used to print values of variables and expressions while you are debugging a program.

14. Invoke the Watch1 window. The Watch window enables you to evaluate variables and expressions. Select the variable `intFac` from the code and drag and drop it in the Watch1 window. You can also double-click the next available row and add a variable to it. Add the variables `i` and `intNumber` to the Watch1 window, as shown in Figure 12.22.


Watch 1

Name	Value	Type
intFac	1	int
i	0	int
intNumber	5	int

FIGURE 12.22
The Watch window enables you to evaluate variables and expressions.

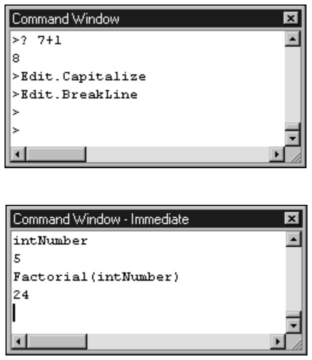
15. Step through the execution of the program by pressing the F11 key. Keep observing the way values change in the Watch1 (or Autos or Locals) window. After a few steps, the method terminates. Note that the program executed only until the value of `i` was `4` and that the loop was not iterated back when the value of `i` was `5`. This causes the incorrect output in the program.

16. Change the condition in the `for` loop to use the `<=` operator instead of `<` and press F11 to step through. The Unable to Apply Code Changes dialog box appears, as shown in Figure 12.23. This dialog box appears because after you have identified the problem and corrected the code, the source code is different from the compiled version of the program. If you choose to continue at this stage, your source code and program in execution are different, and that might mislead you. I recommend that you always restart execution in this case by clicking the Restart button. The code is then recompiled, and the program is started again.

NOTE

Two Modes of the Command Window
The command window has two modes: the command mode and the immediate mode. When you select View, Other Windows, Command Window, the command window is invoked in the command mode. You can distinctly identify the command mode as in this mode the command window shows the > prompt (see Figure 12.21). You can use the command mode to evaluate expressions or to issue commands such as Edit to edit text in a file. You can also use regular expressions with the Edit command to make editing operations quick and effective.

On the other hand, when you invoke the command window by selecting Debug, Window, Immediate, it opens in the immediate mode. You can use the immediate mode to evaluate expressions in the currently debugged program. The immediate mode does not show any prompt (see Figure 12.21). You can switch from immediate mode to command mode by typing `>cmd`, and you can switch from command mode to immediate mode by typing `immed` in the Command window.

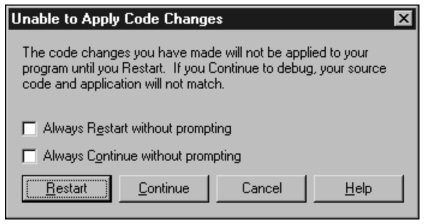

Command Window

```
>? 7+1
8
>Edit.Capitalize
>Edit.BreakLine
>
>
```

Command Window - Immediate

```
intNumber
5
Factorial(intNumber)
24
```

FIGURE 12.21
The Command window can appear in two modes: the command mode and the immediate mode.



Unable to Apply Code Changes

The code changes you have made will not be applied to your program until you Restart. If you Continue to debug, your source code and application will not match.

☐ Always Restart without prompting

☐ Always Continue without prompting

FIGURE 12.23
The Unable to Apply Code Changes dialog box appears if you edit code and then try to continue execution.

17. Enter the value `5` and click the Continue button. The program breaks into the debugger again because the breakpoint is still active. Step through the program and watch the values of the variables. The loop is executed the correct number of times, and you get the correct factorial value.

NOTE

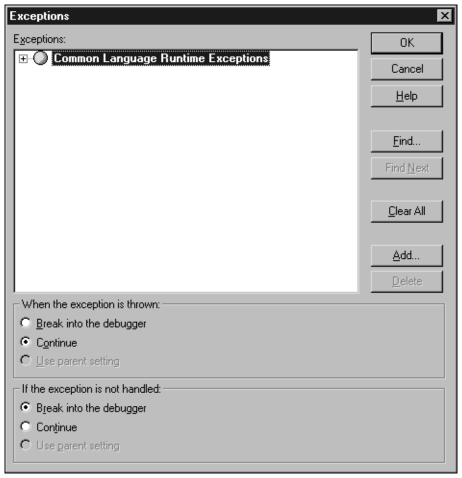
Support for Cross-Language Debugging Visual Studio .NET supports debugging of projects that contain code written in several managed languages. The debugger can transparently step into and out of languages, making the debugging process smooth for you as a developer. Visual Studio .NET also extends this support to nonmanaged languages, but with minor limitations.

Debugging on Exceptions

You can control the way the debugger behaves when it encounters a line of code that throws an exception. You can control this behavior through the Exceptions dialog box, which is shown in Figure 12.24 and is invoked by selecting Debug, Exceptions. The Exceptions dialog box allows you to control the debugger's behavior for each type of exception defined on the system. In fact, if you have defined your own exceptions, you can also add them to this dialog box.

There are two levels at which you can control the behavior of the debugger when it encounters exceptions:

- **When the exception is thrown**—You can instruct the debugger to either continue or break the execution of the program when an exception is thrown. The default setting for Common Language Runtime (CLR) exceptions is to continue the execution, possibly in anticipation that there will be an exception handler.
- **If the exception is not handled**—If the program you are debugging fails to handle an exception, you can instruct the debugger to either ignore it and continue or to break the execution of the program. The default setting for CLR exceptions is to break the execution, warning the programmer of the possibly problematic situation.



Exceptions

Exceptions:
Common Language Runtime Exceptions

When the exception is thrown:
Break into the debugger
[✓ Continue]
Use parent setting

If the exception is not handled:
[✓ Break into the debugger]
Continue
Use parent setting

FIGURE 12.24
The Exceptions dialog box allows you to control the debugger's behavior for system and custom-defined exceptions.

GUIDED PRACTICE EXERCISE 12.2

The Factorial Calculator program created in Step by Step 12.4 throws exceptions of type `System.FormatException` and `System.OverflowException` when users are not careful about the numbers they enter.

The later versions of this program (created in Step by Step 12.6 and 12.7) catch the exception to prevent users from complaining about the annoying exception messages.

The goal of this exercise is to configure the debugger in Step by Step 12.7 so that when the reported exception occurs, you get an opportunity to analyze the program.

How would you configure the debugger?

In this exercise you will practice configuring the exception handling for the Visual Studio .NET debugger environment. You should try working through this problem on your own first. If you get stuck, or if you'd like to see one possible solution, follow these steps:

1. Open the Windows application project `StepByStep12_7`.
2. Activate the Exceptions dialog box by selecting Debug – Exceptions.
3. In the Exceptions dialog box, click the Find button. Enter `System.FormatException` and click the OK button. You are quickly taken to the desired exception in the exception tree view.
4. Select Break into the Debugger from the When the Exception Is Thrown group box.
5. Repeat the steps 3 and 4 for `System.OverFlowException`.
6. Run the project. Enter a nonnumeric value for which to find the factorial. This causes a `System.FormatException` error, and the debugger prompts you to either break or continue the execution. Select to break. You can see the values of various variables at this stage either by moving the mouse pointer over them or by adding the variables to the Watch window. On the next execution of the program, enter a very large value. This causes a `System.OverFlowException` error. Select to break when prompted by the debugger, and then analyze the values of the various variables.

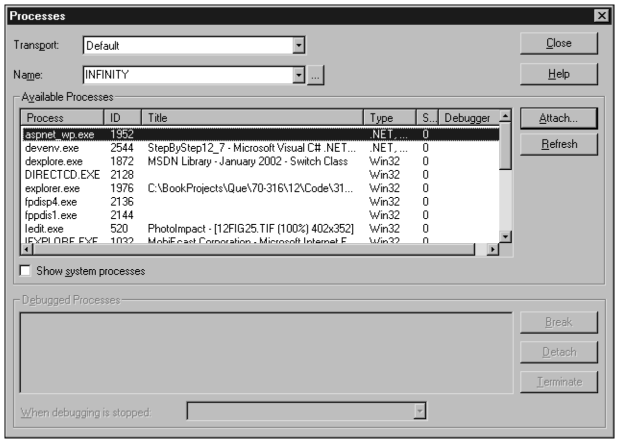
If you have difficulty following this exercise, review the section "Debugging on Exceptions," earlier in this chapter. After doing that review, try this exercise again.

Debugging a Running Process

Until this point in the chapter, you have only seen how to debug programs by starting them from the Visual Studio .NET environment. However, Visual Studio .NET also allows you to debug processes that are running outside the Visual Studio .NET debugging environment.

To access external processes from Visual Studio .NET, you need to invoke the Processes dialog box, shown in Figure 12.25. You can do this in two ways:

- When you have a solution open in Visual Studio .NET, you can invoke the Processes dialog box by selecting Debug, Processes.
- When there is no solution open in Visual Studio .NET, you don't see any Debug menu, but you can still invoke the Processes dialog box by selecting Tools, Debug Processes.



Transport: Default

Name: INFINITY

Available Processes

Process	ID	Title	Type	S...	Debugger
aspnet_wp.exe	1952		.NET, ...	0	
devenv.exe	2544	StepByStep12_7 - Microsoft Visual C# .NET...	.NET, ...	0	
dexplore.exe	1872	MSDN Library - January 2002 - Switch Class	Win32	0	
DIRECTCD.EXE	2128		Win32	0	
explorer.exe	1976	C:\BookProjects\Que\70-316\12\Code\31...	Win32	0	
fpdisp4.exe	2136		Win32	0	
fppdis1.exe	2144		Win32	0	
ledit.exe	520	PhotoImpact - [12FIG25.TIF (100%) 402x352]	Win32	0	
IEXPLORE.EXE	1032	MobiFcast Corporation - Microsoft Internet E	Win32	0	

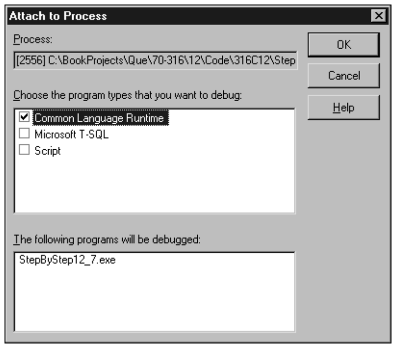
☐ Show system processes

Debugged Processes

When debugging is stopped:

FIGURE 12.25
The Processes dialog box allows you to attach a debugger to a process that is under execution.

Step by Step 12.8 demonstrates how to attach the debugger to a process that is being executed.



Process:

[2556] C:\BookProjects\Que\70-316\12\Code\316C12\Step

Choose the program types that you want to debug:

- [✓] Common Language Runtime
- [ ] Microsoft T-SQL
- [ ] Script

The following programs will be debugged:

StepByStep12_7.exe

FIGURE 12.26
The Attach to Process dialog box allows you to attach to a program that is running in a process outside Visual Studio .NET.

STEP BY STEP

12.8 Attaching the Debugger to a Process That Is Being Executed

1. Using Windows Explorer, navigate to the `bin\Debug` folder inside the project folder for `StepByStep12_7`. Double-click the `.exe` file to launch the program.
2. Start a new instance of Visual Studio .NET and select Tools, Debug Processes. The Processes dialog box appears, as shown in Figure 12.25. You may have a different process list from what is shown in the figure.
3. Select the process named `StepByStep12_7.exe` and click the Attach button. This invokes an Attach to Process dialog box, as shown in Figure 12.26. Select the Common Language Runtime as the program type and keep all other options unchecked. Click the OK button. You should now see the selected process in the Debugged Processes section of the Processes dialog box.
4. Click the Break button to break into the running process. Click the Close button to close the Processes dialog box for now.
5. Both the Disassembly window and the source code window open in the debugging environment. Switch to the source code window. Set a breakpoint on the line of code that makes a call to the `Factorial()` method. Press F11 to step into the program.
6. Enter the value `5` in the form and click the Calculate button. The debugger breaks the execution when the breakpoint is reached.
7. Select Watch, Locals, Autos to analyze variables and step through the program execution.
8. When the factorial result is displayed, invoke the Processes window again by selecting Debug, Processes. From the list of debugged processes, select `StepByStep12_7` and click the Detach button.

9. Click the Close button to close the Processes dialog box. `StepByStep12_7.exe` is still executing, as it was when you initiated the debugging process.

Debugging a Remote Process

The process of debugging a remote process is almost the same as the process of debugging an already running process. The only difference is that prior to selecting a running process from the Processes dialog box, you need to select the remote machine name from the Processes dialog box (refer to Figure 12.25).

Before you can remotely debug processes, you need to do a one-time configuration on the remote machine (where the processes are running). To do so, you take one of the following steps:

- Install Visual Studio .NET on the remote machine.
- Install Remote Components Setup on the remote machine (you can start this from the Visual Studio .NET Setup Disc 1).

Using either of these methods, you can set up Machine Debug Manager (`mdm.exe`) on the remote computer. `mdm.exe` runs as a background service on the computer, providing remote debugging support. In addition, when you use either of these methods, you can add the logged-on user to the Debugger Users group. A user needs to be a member of this group in order to remotely access this computer. You can later add other usernames to this group by using the Computer Management MMC Snap-in on the remote computer.

EXAM TIP

Debugging a Remote Process
The local computer and the remote computer must be members of a trusted domain in order for remote debugging to be possible.

If SQL Server is installed on the remote machine, the setup process just described also configures the machine for SQL Server stored procedures debugging, which is demonstrated at the end of this chapter, in Exercise 12.2.

For a different configuration or requirement, you might want to refer to the “Setting Up Remote Debugging” topic in the Visual Studio .NET Combined Help Collection.

Debugging the Code in DLL Files

The process of debugging a DLL file is similar to the process of debugging an EXE file. There is one difference though: The code in the DLL file cannot be directly invoked, so you need to have a calling program that calls various methods/components of the DLL files.

You typically need to take the following steps in order to debug code in a DLL file:

1. Launch the EXE file that uses the components or methods in the DLL file.
2. Launch Visual Studio .NET and attach the debugger to the EXE file. Set a breakpoint where the method in the DLL file is called. Continue with the execution.
3. The execution breaks when the breakpoint is reached. At this point, select Debug, Step Into to step into the code of the DLL file. Execute the code in the DLL file in step mode while you watch the value of its variables.

In addition, if the code files are executing on a remote machine, you need to make sure that the remote machine is set up with remote debugging support, as explained in the previous section.

REVIEW BREAK

- Debugging is the process of finding the causes of errors in a program, locating the lines of code that are causing the error, and fixing the errors.
- The three options available while performing step-by-step execution are Step Into, Step Over, and Step Out.
- Breakpoints allow you to mark code that signals the debugger to pause execution. After you encounter a breakpoint, you can choose to continue step-by-step execution or resume the normal execution by pressing F5 or by clicking the Resume button, or the Continue button.
- The various tool windows, such as This, Locals, Immediate, Autos, Watch, and Call Stack, can be of great help in tracking the execution path and the status of variables in the process of debugging an application in Visual Studio .NET.

- When an exception is thrown by an application, you can either choose to continue execution or break into the debugger (in order to start debugging operations such as step-by-step execution). You can customize this behavior for each exception object by using the Exceptions dialog box.
- You can attach the debugger to a running process (either local or remote) with the help of the Processes dialog box.

CHAPTER SUMMARY

This chapter starts with a discussion of the various types of tests and how important testing is for an application. You have learned that designing and executing a comprehensive test plan is desirable to ensure that an application is robust, accurate, and reliable.

The .NET Framework provides various classes and techniques that implement tracing in applications. You use tracing to display informative messages during execution of a program. The `Trace` and `Debug` classes provide different methods to generate messages at specific locations in the code. You have learned how trace switches can be applied to an application to give you control over the type of tracing information generated by an application without even needing to recompile the application.

You have also learned about the various C# preprocessing directives that are available in Visual C# .NET. You have seen how you can use the `Conditional` attribute to conditionally compile methods.

The compiler flags syntactical errors at compile time. The tough job is to find logical and runtime errors in an application. Visual C# .NET offers lots of tools for debugging. In this chapter you have learned about various tools available for debugging. You have also learned how to debug an already running process, debug a process running on a remote machine and debug DLL files. As you continue to work with Visual C# .NET, you'll discover more benefits of these debugging tools.

KEY TERMS

- debugging
- testing
- tracing

APPLY YOUR KNOWLEDGE

Exercises

12.1 Creating a Custom Trace Switch

The `TraceSwitch` and `BooleanSwitch` classes are two classes that provide trace switch functionality. If you need different trace levels or different implementations of the `Switch` class, you can inherit from the `Switch` class to implement your own custom trace switches.

In this exercise, you will learn how to create a custom switch. You will create a `FactorialSwitch` class that can be set with four values (`Negative (-1)`, `Off (0)`, `Overflow (1)`, and `Both (2)`) for the Factorial Calculator form. The class will have two properties: `Negative` and `Overflow`.

Estimated time: 25 minutes

1. Launch Visual Studio .NET. Select File, New, Blank Solution, and name the new project `316C12Exercises`.

2. Add a new Windows application project to the solution. Name the project `Exercise12_1`.

3. Using the Add Class Wizard, add a new class to the project. Name the class `FactorialSwitch` and modify the class definition so that it has the following code:

```
using System;
using System.Diagnostics;

namespace Exercise12_1
{
    // The possible values for new switch
    public enum FactorialSwitchLevel
    {
        Negative    = -1,
        Off         =  0,
        Overflow    =  1,
        Both        =  2
    }
    public class FactorialSwitch : Switch
    {
        public FactorialSwitch(
           string displayName,
           string description)
           : base(displayName, description)
        {
        }
        public bool Negative
        {
            get
            {
                // return true if the
                // SwitchSetting is
                // Negative or Both
                if( (SwitchSetting == -1) ||
                   (SwitchSetting == 2))
                    return true;
                else
                    return false;
            }
        }
        public bool Overflow
        {
            get
            {
                // return true if the
                // SwitchSetting is
                // Overflow or Both
                if ((SwitchSetting == 1) ||
                    (SwitchSetting == 2))
                    return true;
                else
                    return false;
            }
        }
    }
}
```

4. In Solution Explorer, right-click `Form1.cs` and rename it `FactorialCalculator`. Open the Properties window for the form and change its `Name` property to `FactorialCalculator` and `Text` property to `Factorial Calculator Exercise 12_1`. Switch to the code view of the form and modify the `Main()` method to launch `FactorialCalculator` instead of `Form1`.

5. Place two TextBox controls (txtNumber and txtFactorial), three Label controls, and a Button control (btnCalculate) on the form and arrange the controls as shown in Figure 12.5.

6. Open FactorialCalculator.cs in the code view. Add the following code in the class definition:

```
static FactorialSwitch facSwitch =
    new FactorialSwitch("FactorialTrace",
    "Trace the factorial application " +
    "using Factorial Switch");
```

7. Attach a Click event handler to the btnCalculate control with the following code:

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
    int intNumber = Convert.ToInt32(
        txtNumber.Text);

    if (facSwitch.Negative)
    {
        // make a debug assertion
        Debug.Assert(
           intNumber >= 0, "Invalid value",
           "negative value in debug mode");
    }

    int intFac = 1;
    for (int i = 2; i <= intNumber; i++)
    {
        intFac = intFac * i;
    }

    if (facSwitch.Overflow)
        // write a debug message if
        // the condition is true
        Debug.WriteLineIf(intFac < 1,
           "There was an overflow",
           "Factorial Program Debug");
    txtFactorial.Text = intFac.ToString();
}
```

8. In Solution Explorer, select View All Files from the toolbar. Navigate to the bin\debug folder. Right-click the debug folder and select Add, Add New Item. Choose to create an XML file and name it Exercise12_1.exe.config.

9. In the XML editor, type the following configuration data in the XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.diagnostics>
        <switches>
            <add name="FactorialTrace"
                 value="2" />
        </switches>
    </system.diagnostics>
</configuration>
```

10. Set Exercise12_1 as the startup project.

11. Run the project, using the default Debug configuration. Notice that the Assertion Failed dialog box is displayed only if the switch is set with the value -1 or 2. Similarly, the overflow message is displayed in the Output window only if the switch value is set to 1 or 2.

12. Modify the program to change all Debug statements to Trace statements. Copy the XML configuration file to the bin\Release folder in the project and then repeat step 11, using the Release configuration.

The value set in the configuration file can be accessed through the SwitchSetting property of the Switch class. The Negative and Overflow properties of the FactorialSwitch class return true or false, depending on the value of the SwitchSetting property.

12.2 Debugging SQL Server Stored Procedures Using Visual C# .NET

You can perform step-by-step execution of SQL Server stored procedures in Visual C# .NET. This exercise shows you how.

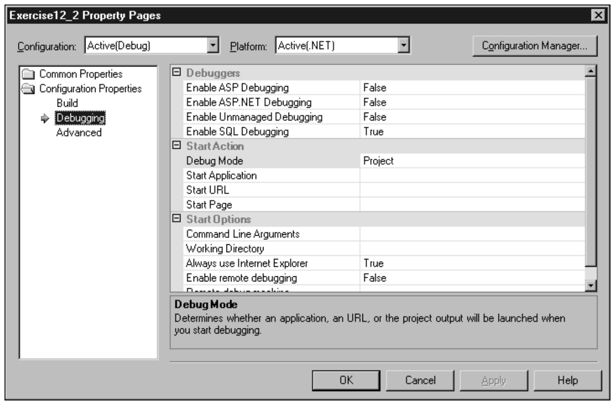
Estimated time: 30 minutes

1. Add a new Windows application project to the solution. Name the project Exercise12_2.

APPLY YOUR KNOWLEDGE

2. Rename the `Form1.cs` form `MostExpensiveProdcuts.cs`. Change all occurrences of `Form1.cs` to `MostExpensiveProducts.cs`.

3. Select Project, Properties from the main menu. Select `Debugging` under the `Configuration Properties` node in the left pane of the project's Property Pages dialog box. In the right pane, under the `Debuggers` node, choose `True` for `Enable SQL Debugging`, as shown in Figure 12.27.



Configuration: Active(Debug) Platform: Active(.NET)

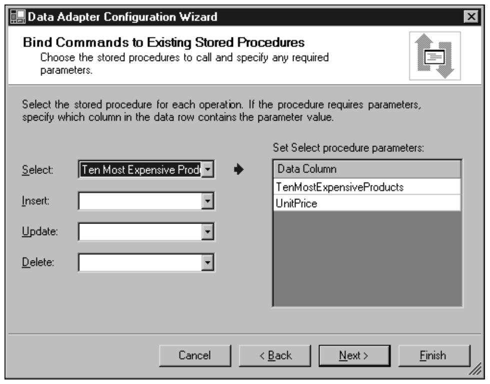
Debuggers	
Enable ASP Debugging	False
Enable ASP.NET Debugging	False
Enable Unmanaged Debugging	False
Enable SQL Debugging	True
Start Action	
Debug Mode	Project
Start Application	
Start URL	
Start Page	
Start Options	
Command Line Arguments	
Working Directory	
Always use Internet Explorer	True
Enable remote debugging	False

Debug Mode
Determines whether an application, an URL, or the project output will be launched when you start debugging.

FIGURE 12.27
You can enable SQL debugging in the project's Property Pages dialog box to allow debugging of SQL Server stored procedures.

4. Drag a `SqlDataAdapter` component to the form. This activates the Data Adapter Configuration Wizard. Click Next. Select the Northwind database connection that you have created in the earlier chapters (Chapters 5, "Data Binding," and 6, "Consuming and Manipulating Data,") or click the New Connection button to create a Northwind database connection. Click Next.

5. Choose the Use Existing Stored Procedures option in the Choose a Query Type page. Click Next. Select Ten Most Expensive Products from the Select combo box, as shown in Figure 12.28. Click Next and then click Finish. A `SqlConnection` component is created in the component tray.



Bind Commands to Existing Stored Procedures
Choose the stored procedures to call and specify any required parameters.

Select the stored procedure for each operation. If the procedure requires parameters, specify which column in the data row contains the parameter value.

Select: Ten Most Expensive Prod

Set Select procedure parameters:

Data Column
TenMostExpensiveProducts
UnitPrice

FIGURE 12.28
The Bind Commands to Existing Stored Procedures dialog box allows you to choose the SQL stored procedures to bind to the `SqlCommand` object.

6. Select the `sqlDataAdapter1` component, and then right-click and select Generate DataSet from the context menu. Select the New radio button and choose Ten Most Expensive Products from the checked list box. Click OK to create a `dataSet11` component in the component tray.

7. Place a `Button` control (`btnGetProducts`) and a `DataGrid` control (`dataGrid1`) on the form. Change the `DataGrid` control's `DataSource` property to `dataSet1` and `DataMember` property to `Ten Most Expensive Products`.

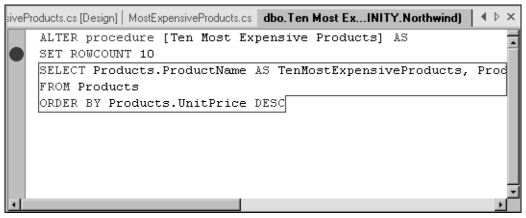
APPLY YOUR KNOWLEDGE

8. Add the following code in the `Click` event of the `Button` control:

```
private void btnGetProducts_Click(
   object sender, System.EventArgs e)
{
    sqlDataAdapter1.Fill(this.dataSet11);
}
```

9. Insert a breakpoint in the `Click` event handler, at the point of a call to the `Fill()` method of the `sqlDataAdapter1` object.

10. Open Server Explorer. Open the `Data Connections` node and select the stored procedure Ten Most Expensive Products. Right-click the stored procedure and select Edit Stored Procedure. Insert a breakpoint in the starting code line of the stored procedure, as shown in Figure 12.29.



```
ALTER procedure [Ten Most Expensive Products] AS
SET ROWCOUNT 10
SELECT Products.ProductName AS TenMostExpensiveProducts, Proc
FROM Products
ORDER BY Products.UnitPrice DESC
```

FIGURE 12.29
You can insert breakpoints in SQL Server stored procedures.

11. Run the project. Click the button. The program starts step-by-step execution as soon as it encounters the breakpoint in the `Fill()` method call line. Press F11. You are taken to the stored procedure code, where you can perform step-by-step execution.

This exercise teaches you how to debug SQL Server stored procedures by using step-by-step execution.

In Figure 12.29, notice the `SELECT` statement enclosed in a blue outline, which represents each step in the stored procedure (if the step occupies more than a line).

EXAM TIP

Watching SQL Server Variables
You can use various tools, such as the Watch and Locals windows, to keep track of the values of the variables that are defined in the stored procedures during step-by-step execution. These tools are very helpful when you are debugging complex stored procedures.

12.3 Setting Conditional Breakpoints by Using Visual C# .NET

This exercise shows you how to set conditional breakpoints. You set a breakpoint in the factorial calculation to break when the factorial value overflows (that is, when it becomes negative).

Estimated time: 30 minutes

1. Add a new Windows application project to the solution. Name the project `Exercise12_3`.

2. In Solution Explorer, right-click `Form1.cs` and rename it `FactorialCalculator`. Open the Properties window for this form and change its `Name` property to `FactorialCalculator` and Text property to `Factorial Calculator Exercise 12_3`. Switch to the code view of the form and modify the `Main()` method to launch `FactorialCalculator` instead of `Form1`.

3. Place two `TextBox` controls (`txtNumber` and `txtFactorial`), three `Label` controls, and a `Button` control (`btnCalculate`) on the form and arrange the controls as shown in Figure 12.5.

APPLY YOUR KNOWLEDGE

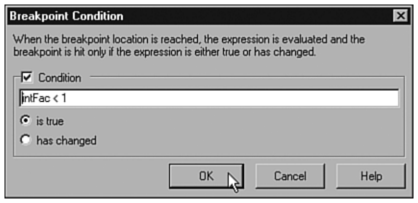
4. Attach a `Click` event handler to the `btnCalculate` control and add the following code in the event handler:

```
private void btnCalculate_Click(
    object sender, System.EventArgs e)
{
    int intNumber =
        Convert.ToInt32(txtNumber.Text);

    int intFac = 1;
    for (int i = 2; i <= intNumber; i++)
    {
        intFac = intFac * i;
    }
    txtFactorial.Text = intFac.ToString();
}
```

5. Right-click the following line in the button `Click` event handler and select New Breakpoint from the context menu:

```
intFac=intFac*i;
```

The New Breakpoint dialog box appears. Select the File tab. Note that File, Line, and Character position are already marked correctly. Click the Condition button. This opens the Breakpoint Condition dialog box. Set the values in the dialog box as shown in Figure 12.30. Select the Condition checkbox. Enter `intFac < 1` in the Condition text box and select the Is True option. Click OK twice to dismiss the New Breakpoint dialog box.


Breakpoint Condition

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is either true or has changed.

[✓] Condition

intFac < 1

(●) is true

( ) has changed

FIGURE 12.30
You can set conditional breakpoints with the Breakpoint Condition dialog box.

6. Run the project using the default `Debug` configuration. Enter `100` and click the Calculate button. Notice that the running page breaks into the debugger when `intFac` has a negative value and the breakpoint is reached.

Review Questions

1. For what do you use a test plan?
2. What is the purpose of the `Assert()` method in the `Debug` and `Trace` classes?
3. What is the main purpose of `TraceListener` class? What classes implement `TraceListener` in the Framework Class Library?
4. What are the two built-in trace switches in the .NET Framework Class Library?
5. What is the main advantage of trace switches?
6. What types of methods can be marked with the `Conditional` attribute?
7. What are the purposes of the `#error` and `#warning` preprocessing directives?
8. What are the three commands can you use to step through code while debugging?
9. What happens when you put a breakpoint in code?
10. What are some of the different windows that are available for debugging?
11. How can you attach the debugger to a running process in Visual C# .NET?
12. In order to verify that remote debugging is enabled on a system, what should you check?

Programming fundamentals :create and use variables and constants. It then goes on to introduce enumerations, strings, identifiers, expressions, and statements.

The second part of the chapter explains and demonstrates the use of branching, using the `if`, `switch`, `while`, `do...while`, `for`, and `foreach` statements. Also discussed are operators, including the assignment, logical, relational, and mathematical operators. This is followed by an introduction to namespaces and a short tutorial on the C# precompiler.

Although C# is principally concerned with the creation and manipulation of objects, it is best to start with the fundamental building blocks: the elements from which objects are created. These include the built-in types that are an intrinsic part of the C# language as well as the syntactic elements of C#.

3.1 Types

C# is a strongly typed language. In a strongly typed language you must declare the type of each object you create (e.g., integers, floats, strings, windows, buttons, etc.) and the compiler will help you prevent bugs by enforcing that only data of the right type is assigned to those objects. The type of an object signals to the compiler the size of that object (e.g., `int` indicates an object of 4 bytes) and its capabilities (e.g., buttons can be drawn, pressed, and so forth).

Like C++ and Java, C# divides types into two sets: *intrinsic* (built-in) types that the language offers and *user-defined* types that the programmer defines.

C# also divides the set of types into two other categories: *value* types and *reference* types.[1] The principal difference between value and reference types is the manner in which their values are stored in memory. A value type holds its actual value in memory allocated on the stack (or it is allocated as part of a larger reference type object). The address of a reference type variable sits on the stack, but the actual object is stored on the heap.

[1] All the intrinsic types are value types except for `Object` (discussed in Chapter 5) and `String` (discussed in Chapter 10). All user-defined types are reference types except for structs (discussed in Chapter 7).

If you have a very large object, putting it on the heap has many advantages. Chapter 4 discusses the various advantages and disadvantages of working with reference types; the current chapter focuses on the intrinsic value types available in C#.

C# also supports C++ style *pointer* types, but these are rarely used, and only when working with unmanaged code. Unmanaged code is code created outside of the .NET platform, such as COM objects. Working with COM objects is discussed in Chapter 22.

3.1.1 Working with Built-in Types

The C# language offers the usual cornucopia of intrinsic (built-in) types one expects in a modern language, each of which maps to an underlying type supported by the .NET Common Language Specification (CLS). Mapping the C# primitive types to the underlying .NET type ensures that objects created in C# can be used interchangeably with objects created in any other language compliant with the .NET CLS, such as VB .NET.

Each type has a specific and unchanging size. Unlike with C++, a C# `int` is always 4 bytes because it maps to an `Int32` in the .NET CLS. Table 3-1 lists the built-in value types offered by C#.

Table 3-1, C# built-in value types

Type	Size (in bytes)	.NET Type	Description
`byte`	1	`Byte`	Unsigned (values 0-255).
`char`	1	`Char`	Unicode characters.
`bool`	1	`Boolean`	`true` or `false`.
`sbyte`	1	`Sbyte`	Signed (values -128 to 127).
`short`	2	`Int16`	Signed (short) (values -32,768 to 32,767).
`ushort`	2	`Uint16`	Unsigned (short) (values 0 to 65,535).
`int`	4	`Int32`	Signed integer values between -2,147,483,647 and 2,147,483,647.
`uint`	4	`Uint32`	Unsigned integer values between 0 and 4,294,967,295.
`float`	4	`Single`	Floating point number. Holds the values from approximately +/-1.5 * 10^{-45} to approximate +/-3.4 * 10^{38} with 7 significant figures.
`double`	8	`Double`	Double-precision floating point; holds the values from approximately +/-5.0 * 10^{-324} to approximate +/-1.7 * 10^{308} with 15-16 significant figures.
`decimal`	8	`Decimal`	Fixed-precision up to 28 digits and the position of the decimal point. This is typically used in financial calculations. Requires the suffix "m" or "M."
`long`	8	`Int64`	Signed integers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`ulong`	8	`Uint64`	Unsigned integers ranging from 0 to 0xffffffffffffffff.

C and C++ programmers take note: Boolean variables can only have the values `true` or `false`. Integer values do not equate to Boolean values in C# and there is no implicit conversion.

In addition to these primitive types, C# has two other value types: `enum` (considered later in this chapter) and `struct` (see Chapter 4). Chapter 4 also discusses other subtleties of value types such as forcing value types to act as reference types through a process known as *boxing*, and that value types do not "inherit."

The Stack and the Heap

A stack is a data structure used to store items on a last-in first-out basis (like a stack of dishes at the buffet line in a restaurant). *The* stack refers to an area of memory supported by the processor, on which the local variables are stored.

In C#, value types (e.g., integers) are allocated on the stack—an area of memory is set aside for their value, and this area is referred to by the name of the variable.

Reference types (e.g., objects) are allocated on the heap. When an object is allocated on the heap its address is returned, and that address is assigned to a reference.

The garbage collector destroys objects on the stack sometime after the stack frame they are declared within ends. Typically a stack frame is defined by a function. Thus, if you declare a local variable within a function (as explained later in this chapter) the object will be marked for garbage collection after the function ends.

Objects on the heap are garbage collected sometime after the final reference to them is destroyed.

3.1.1.1 Choosing a built-in type

Typically you decide which size integer to use (`short`, `int`, or `long`) based on the magnitude of the value you want to store. For example, a `ushort` can only hold values from 0 through 65,535, while a `ulong` can hold values from 0 through 4,294,967,295.

That said, memory is fairly cheap, and programmer time is increasingly expensive; most of the time you'll simply declare your variables to be of type `int`, unless there is a good reason to do otherwise.

The signed types are the numeric types of choice of most programmers unless they have a good reason to use an unsigned value.

Although you might be tempted to use an unsigned `short` to double the positive values of a signed `short` (moving the maximum positive value from 32,767 up to 65,535), it is easier and preferable to use a signed integer (with a maximum value of 2,147,483,647).

It is better to use an unsigned variable when the fact that the value must be positive is an inherent characteristic of the data. For example, if you had a variable to hold a person's age, you would use an unsigned `int` because an age cannot be negative.

`Float`, `double`, and `decimal` offer varying degrees of size and precision. For most small fractional numbers, `float` is fine. Note that the compiler assumes that any number with a decimal point is a double unless you tell it otherwise. To assign a literal `float`, follow the number with the letter `f`. (Assigning values to literals is discussed in detail later in this chapter.)

```
float someFloat = 57f;
```

The `char` type represents a Unicode character. `char` literals can be simple, Unicode, or escape characters enclosed by single quote marks. For example, `A` is a simple character while `\u0041` is a Unicode character. Escape characters are special two-character tokens in which the first character is a backslash. For example, `\t` is a horizontal tab. The common escape characters are shown in Table 3-2.

Table 3-2, Common escape characters

Char	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\0`	Null
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

3.1.1.2 Converting built-in types

Objects of one type can be converted into objects of another type either implicitly or explicitly. Implicit conversions happen automatically; the compiler takes care of it for you. Explicit conversions happen when you "cast" a value to a different type. The semantics of an explicit conversion are "Hey! Compiler! I know what I'm doing." This is sometimes called "hitting it with the big hammer" and can be very useful or very painful, depending on whether your thumb is in the way of the nail.

Implicit conversions happen automatically and are guaranteed not to lose information. For example, you can implicitly cast from a `short int` (2 bytes) to an `int` (4 bytes) implicitly. No matter what value is in the `short`, it will not be lost when converting to an `int`:

```
short x = 5;
int y = x; // implicit conversion
```

If you convert the other way, however, you certainly can lose information. If the value in the `int` is greater than 32,767 it will be truncated in the conversion. The compiler will not perform an implicit conversion from `int` to `short`:

```
short x;
int y = 500;
x = y;  // won't compile
```

You must explicitly convert using the cast operator:

```
short x;
int y = 500;
x = (short) y;  // OK
```

All of the intrinsic types define their own conversion rules. At times it is convenient to define conversion rules for your user-defined types, as discussed in Chapter 5.

3.2 Variables and Constants

A variable is a storage location with a type. In the preceding examples, both *x* and *y* are variables. Variables can have values assigned to them, and that value can be changed programmatically.

WriteLine( )

The .Net Framework provides a useful method for writing output to the screen. The details of this method, `System.Console.WriteLine( )`, will become clearer as we progress through the book, but the fundamentals are straightforward. You call the method as shown in Example 3-3, passing in a string that you want printed to the console (the screen) and, optionally, parameters that will be substituted. In the following example:

```
System.Console.WriteLine("After assignment, myInt: {0}", myInt);
```

the string "`After assignment, myInt:`" is printed as is, followed by the value in the variable `myInt`. The location of the *substitution parameter* `{0}` specifies where the value of the first output variable, `myInt`, will be displayed, in this case at the end of the string. We'll see a great deal more about `Writeline( )` in coming chapters.

You create a variable by declaring its type and then giving it a name. You can initialize the variable when you declare it, and you can assign a new value to that variable at any time, changing the value held in the variable. This is illustrated in Example 3-1.

Example 3-1. Initializing and assigning a value to a variable

```
class Values
{
   static void Main(  )
   {
      int myInt = 7;
      System.Console.WriteLine("Initialized, myInt: {0}",
         myInt);
      myInt = 5;
      System.Console.WriteLine("After assignment, myInt: {0}",
         myInt);
   }
}

 Output:
Initialized, myInt: 7
After assignment, myInt: 5
```

Here we initialize the variable `myInt` to the value `7`, display that value, reassign the variable with the value `5`, and display it again.

3.2.1 Definite Assignment

C# requires that variables be initialized before they are used. To test this rule, change the line that initializes `myInt` in Example 3-1 to:

```
int myInt;
```

and save the revised program shown in Example 3-2.

Example 3-2. Using an uninitialized variable

```
class Values
{
   static void Main(  )
   {
      int myInt;
      System.Console.WriteLine
      ("Uninitialized, myInt: {0}",myInt);
      myInt = 5;
      System.Console.WriteLine("Assigned, myInt: {0}", myInt);
   }
}
```

When you try to compile this listing, the C# compiler will display the following error message:

```
3.1.cs(6,55): error CS0165: Use of unassigned local
variable 'myInt'
```

It is not legal to use an uninitialized variable in C#. Example 3-2 is not legal.

So, does this mean you must initialize every variable in a program? In fact, no. You don't actually need to initialize a variable, but you must assign a value to it before you attempt to use it. This is called *definite assignment and C# requires it.* Example 3-3 illustrates a correct program.

Example 3-3. Assigning without initializing

```
class Values
{
   static void Main(  )
   {
      int myInt;
      myInt = 7;
      System.Console.WriteLine("Assigned, myInt: {0}", myInt);
      myInt = 5;
      System.Console.WriteLine("Reassigned, myInt: {0}", myInt);
   }
}
```

3.2.2 Constants

A *constant* is a variable whose value cannot be changed. Variables are a powerful tool, but there are times when you want to manipulate a defined value, one whose value you want to ensure remains constant. For example, you might need to work with the Fahrenheit freezing and boiling points of water in a program simulating a chemistry experiment. Your program will be clearer if you name the variables that store these values `FreezingPoint` and `BoilingPoint`, but you do not want to permit their values to be reassigned. How do you prevent reassignment? The answer is to use a constant.

Constants come in three flavors: *literals*, *symbolic constants,* and *enumerations*. In this assignment:

```
x = 32;
```

the value `32` is a literal constant. The value of 32 is always 32. You can't assign a new value to 32; you can't make 32 represent the value `99` no matter how you might try.

Symbolic constants assign a name to a constant value. You declare a symbolic constant using the `const` keyword and the following syntax:

```
const type identifier = value;
```

A constant must be initialized when it is declared, and once initialized it cannot be altered. For example:

```
const int FreezingPoint = 32;
```

In this declaration, 32 is a literal constant and `FreezingPoint` is a symbolic constant of type `int`. Example 3-4 illustrates the use of symbolic constants.

Example 3-4. Using symbolic constants

```
class Values
{
   static void Main(  )
   {
      const int FreezingPoint = 32;   // degrees Farenheit
      const int BoilingPoint = 212;

      System.Console.WriteLine("Freezing point of water: {0}",
            FreezingPoint );
      System.Console.WriteLine("Boiling point of water: {0}",
            BoilingPoint );
      //BoilingPoint = 21;
   }
}
```

Example 3-4 creates two symbolic integer constants: `FreezingPoint` and `BoilingPoint`. As a matter of style, constant names are written in Pascal notation, but this is certainly not required by the language.

These constants serve the same purpose as always using the *literal* values `32` and `212` for the freezing and boiling points of water in expressions that require them, but because these constants have names they convey far more meaning. Also, if you decide to switch this program to Celsius, you can reinitialize these constants at compile time, to `0` and `100`, respectively; and all the rest of the code ought to continue to work.

To prove to yourself that the constant cannot be reassigned, try uncommenting the last line of the program (shown in bold). When you recompile you should receive this error:

```
error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
```

3.2.3 Enumerations

Enumerations provide a powerful alternative to constants. An enumeration is a distinct value type, consisting of a set of named constants (called the *enumerator list*).

In Example 3-4, you created two related constants:

```
const int FreezingPoint = 32;
const int BoilingPoint = 212;
```

You might wish to add a number of other useful constants as well to this list, such as:

```
const int LightJacketWeather = 60;
const int SwimmingWeather = 72;
const int WickedCold = 0;
```

This process is somewhat cumbersome, and there is no logical connection among these various constants. C# provides the *enumeration* to solve these problems:

```
enum Temperatures
{
   WickedCold = 0,
   FreezingPoint = 32,
   LightJacketWeather = 60,
   SwimmingWeather = 72,
   BoilingPoint = 212,
}
```

Every enumeration has an underlying type, which can be any integral type (`integer`, `short`, `long`, etc.) except for `char`. The technical definition of an enumeration is:

```
[attributes] [modifiers] enum identifier
      [:base-type] {enumerator-list};
```

The optional attributes and modifiers are considered later in this book. For now, let's focus on the rest of this declaration. An enumeration begins with the keyword `enum`, which is generally followed by an identifier, such as:

```
enum Temperatures
```

The base type is the underlying type for the enumeration. If you leave out this optional value (and often you will) it defaults to `integer`, but you are free to use any of the integral types (e.g., `ushort`, `long`) except for `char`. For example, the following fragment declares an enumeration of unsigned integers (`uint`):

```
enum ServingSizes :uint
{
    Small = 1,
    Regular = 2,
    Large = 3
}
```

Notice that an `enum` declaration ends with the enumerator list. The enumerator list contains the constant assignments for the enumeration, each separated by a comma.

Example 3-5 rewrites Example 3-4 to use an enumeration.

Example 3-5. Using enumerations to simplify your code

```
class Values
{

   enum Temperatures
   {
      WickedCold = 0,
      FreezingPoint = 32,
      LightJacketWeather = 60,
      SwimmingWEather = 72,
      BoilingPoint = 212,
   }

   static void Main(  )
   {

      System.Console.WriteLine("Freezing point of water: {0}",
         Temperatures.FreezingPoint );
      System.Console.WriteLine("Boiling point of water: {0}",
         Temperatures.BoilingPoint );

   }
}
```

As you can see, an `enum` must be qualified by its enumtype (e.g., `Temperatures.WickedCold`).

Each constant in an enumeration corresponds to a numerical value, in this case, an integer. If you don't specifically set it otherwise, the enumeration begins at 0 and each subsequent value counts up from the previous.

If you create the following enumeration:

```
enum SomeValues
{
   First,
   Second,
   Third = 20,
   Fourth
}
```

the value of `First` will be `0`, `Second` will be `1`, `Third` will be `20`, and `Fourth` will be 21.

Enums are formal types; therefore an explicit conversion is required to convert between an enum type and an integral type.

> *C++ programmers take note:* C#'s use of enums is subtly different from C++, which restricts assignment to an enum type from an integer but allows an enum to be promoted to an integer for assignment of an enum to an integer.

3.2.4 Strings

It is nearly impossible to write a C# program without creating strings. A string object holds a string of characters.

You declare a string variable using the `string` keyword much as you would create an instance of any object:

```
string myString;
```

A string literal is created by placing double quotes around a string of letters:

```
"Hello World"
```

It is common to initialize a string variable with a string literal:

```
string myString = "Hello World";
```

Strings will be covered in much greater detail in Chapter 10.

3.2.5 Identifiers

Identifiers are names that programmers choose for their types, methods, variables, constants, objects, and so forth. An identifier must begin with a letter or an underscore.

The Microsoft naming conventions suggest using *camel notation* (initial lowercase such as `someName`) for variable names and *Pascal notation* (initial uppercase such as `SomeOtherName`) for method names and most other identifiers.

> Microsoft no longer recommends using Hungarian notation (e.g., `iSomeInteger`) or underscores (e.g., `SOME_VALUE`).

Identifiers cannot clash with keywords. Thus, you cannot create a variable named `int` or `class`. In addition, identifiers are case-sensitive, so C# treats `myVariable` and `MyVariable` as two different variable names.

3.3 Expressions

Statements that evaluate to a value are called *expressions*. You may be surprised how many statements do evaluate to a value. For example, an assignment such as:

```
myVariable = 57;
```

is an expression; it evaluates to the value assigned, in this case, `57`.

Note that the preceding statement assigns the value `57` to the variable `myVariable`. The assignment operator (`=`) does not test equality; rather it causes whatever is on the right side (`57`) to be assigned to whatever is on the left side (`myVariable`). All of the C# operators (including assignment and equality) are discussed later in this chapter (see Section 3.6).

Because `myVariable = 57` is an expression that evaluates to `57`, it can be used as part of another assignment operator, such as:

```
mySecondVariable = myVariable = 57;
```

What happens in this statement is that the literal value `57` is assigned to the variable `myVariable`. The value of that assignment (`57`) is then assigned to the second variable, `mySecondVariable`. Thus, the value `57` is assigned to both variables. You can thus initialize any number of variables to the same value with one statement:

```
a = b = c = d = e = 20;
```

3.4 Whitespace

In the C# language, spaces, tabs, and newlines are considered to be " whitespace" (so named because you see only the white of the underlying "page"). Extra whitespace is generally ignored in C# statements. Thus, you can write:

```
myVariable = 5;
```

or:

```
myVariable    =                             5;
```

and the compiler will treat the two statements as identical.

The exception to this rule is that whitespace within strings is not ignored. If you write:

```
Console.WriteLine("Hello World")
```

each space between "Hello" and "World" is treated as another character in the string.

Most of the time the use of whitespace is intuitive. The key is to use whitespace to make the program more readable to the programmer; the compiler is indifferent.

However, there are instances in which the use of whitespace is quite significant. Although the expression:

```
int x = 5;
```

is the same as:

```
int x=5;
```

it is not the same as:

```
intx=5;
```

The compiler knows that the whitespace on either side of the assignment operator is extra, but the whitespace between the type declaration `int` and the variable name `x` is *not* extra, and is required. This is not surprising; the whitespace allows the compiler to parse the keyword `int` rather than some unknown term `intx`. You are free to add as much or as little whitespace between `int` and `x` as you care to, but there must be at least one whitespace character (typically a space or tab).

> *Visual Basic programmers take note:* in C# the end-of-line has no special significance; statements are ended with semicolons, not newline characters. There is no line continuation character because none is needed.

3.5 Statements

In C# a complete program instruction is called a *statement*. Programs consist of sequences of C# statements. Each statement must end with a semicolon (`;`). For example:

```
int x; // a statement
x = 23; // another statement
int y = x; // yet another statement
```

C# statements are evaluated in order. The compiler starts at the beginning of a statement list and makes its way to the bottom. This would be entirely straightforward, and terribly limiting, were it not for branching. There are two types of branches in a C# program: *unconditional branching* and *conditional branching*.

Program flow is also affected by looping and iteration statements, which are signaled by the keywords `for` , `while`, `do`, `in`, and `foreach`. Iteration is discussed later in this chapter. For now, let's consider some of the more basic methods of conditional and unconditional branching.

3.5.1 Unconditional Branching Statements

An unconditional branch is created in one of two ways. The first way is by invoking a method. When the compiler encounters the name of a method it stops execution in the current method and branches to the newly "called" method. When that method returns a value, execution picks up in the original method on the line just below the method call. Example 3-6 illustrates.

Example 3-6. Calling a method

```
using System;
class Functions
{
    static void Main(  )
    {
        Console.WriteLine("In Main! Calling SomeMethod(  )...");
        SomeMethod(  );
        Console.WriteLine("Back in Main(  ).");

    }
    static void SomeMethod(  )
    {
        Console.WriteLine("Greetings from SomeMethod!");
    }
}

Output:

In Main! Calling SomeMethod(  )...
Greetings from SomeMethod!
Back in Main(  ).
```

Program flow begins in `Main( )` and proceeds until `SomeMethod( )` is invoked (invoking a method is sometimes referred to as "calling" the method). At that point program flow branches to the method. When the method completes, program flow resumes at the next line after the call to that method.

The second way to create an unconditional branch is with one of the unconditional branch keywords: `goto`, `break`, `continue`, `return`, or `statementthrow`. Additional information about the first four jump statements is provided in Section 3.5.2.3, Section 3.5.3.1, and Section 3.5.3.6, later in this chapter. The final statement, `throw`, is discussed in Chapter 9.

3.5.2 Conditional Branching Statements

A conditional branch is created by a conditional statement, which is signaled by keywords such as `if`, `else`, or `switch`. A conditional branch occurs only if the condition expression evaluates true.

3.5.2.1 If...else statements

`If...else` statements branch based on a condition. The condition is an expression, tested in the head of the `if` statement. If the condition evaluates true, the statement (or block of statements) in the body of the `if` statement is executed.

`If` statements may contain an optional `else` statement. The `else` statement is executed only if the expression in the head of the `if` statement evaluates false:

```
if (expression)
   statement1
[else
   statement2]
```

This is the kind of description of the `if` statement you are likely to find in your compiler documentation. It shows you that the `if` statement takes an *expression* (a statement that returns a value) in parentheses, and executes `statement1` if the expression evaluates true. Note that `statement1` can actually be a block of statements within braces.

You can also see that the `else` statement is optional, as it is enclosed in square brackets. Although this gives you the syntax of an `if` statement, an illustration will make its use clear. Example 3-7 illustrates.

Example 3-7. If . . . else statements

```
using System;
class Values
{
   static void Main(  )
   {
      int valueOne = 10;
      int valueTwo = 20;

      if ( valueOne > valueTwo )
      {
         Console.WriteLine(
           "ValueOne: {0} larger than ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         Console.WriteLine(
          "ValueTwo: {0} larger than ValueOne: {1}",
                valueTwo,valueOne);
      }

      valueOne = 30; // set valueOne higher

      if ( valueOne > valueTwo )
      {
         valueTwo = valueOne++;
         Console.WriteLine("\nSetting valueTwo to valueOne value, ");
         Console.WriteLine("and incrementing ValueOne.\n");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         valueOne = valueTwo;
         Console.WriteLine("Setting them equal. ");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
   }
}
```

In Example 3-7, the first `if` statement tests whether `valueOne` is greater than `valueTwo`. The relational operators such as greater than (>), less than (<), and equal to (==) are fairly intuitive to use.

The test of whether `valueOne` is greater than `valueTwo` evaluates false (because `valueOne` is 10 and `valueTwo` is 20 and so `valueOne` is *not* greater than `valueTwo`). The `else` statement is invoked, printing the statement:

```
ValueTwo: 20 is larger than ValueOne: 10
```

The second `if` statement evaluates true and all the statements in the `if` block are evaluated, causing two lines to print:

```
ValueOne was larger. Setting valueTwo to old ValueOne value,
and incrementing ValueOne.

ValueOne: 31  ValueTwo: 30
```

Statement Blocks

Anyplace that C# expects a statement, you can substitute a statement block. A *statement block* is a set of statements surrounded by braces.

Thus, where you might write:

```
if (someCondition)
    someStatement;
```

you can instead write:

```
if(someCondition)
{
    statementOne;
    statementTwo;
    statementThree;
}
```

3.5.2.2 Nested if statements

It is possible, and not uncommon, to nest `if` statements to handle complex conditions. For example, suppose you need to write a program to evaluate the temperature, and specifically to return the following types of information:

- If the temperature is 32 degrees or lower, the program should warn you about ice on the road.
- If the temperature is exactly 32 degrees, the program should tell you that there may be ice patches.
- If the temperature is higher than 32 degrees, the program should assure you that there is no ice.

There are many good ways to write this program. Example 3-8 illustrates one approach, using nested `if` statements.

Example 3-8. Nested if statements

```
using System;
class Values
{
   static void Main(  )
   {
      int temp = 32;

      if (temp <= 32)
      {
         Console.WriteLine("Warning! Ice on road!");
         if (temp == 32)
         {
        Console.WriteLine(
          "Temp exactly freezing, beware of water.");
         }
         else
         {
            Console.WriteLine("Watch for black ice! Temp: {0}", temp);
         }
      }
```

```
    }
}
```

The logic of Example 3-8 is that it tests whether the temperature is less than or equal to 32. If so, it prints a warning:

```
if (temp <= 32)
{
   Console.WriteLine("Warning! Ice on road!");
```

The program then checks whether the temp is equal to 32 degrees. If so, it prints one message; if not, the temp must be less than 32 and the program prints the second message. Notice that this second `if` statement is nested within the first `if`, so the logic of the `else` is: "since it has been established that the temp is less than or equal to 32, and it isn't equal to 32, it must be less than 32."

All Operators Are Not Created Equal

A closer examination of the second `if` statement in Example 3-8 reveals a common potential problem. This `if` statement tests whether the temperature is equal to 32:

```
if (temp == 32)
```

In C and C++, there is an inherent danger in this kind of statement. It's not uncommon for novice programmers to use the assignment operator rather than the equals operator, instead creating the statement:

```
if (temp = 32)
```

This mistake would be difficult to notice, and the result would be that `32` was assigned to `temp`, and then `32` would be returned as the value of the assignment statement. Because any nonzero value evaluates to true in C and C#, the `if` statement would return true. The side effect would be that `temp` would be assigned a value of `32` whether or not it originally had that value. This is a common bug that could easily be overlooked—if the developers of C# had not anticipated it!

C# solves this problem by requiring that `if` statements accept only Boolean values. The `32` returned by the assignment is not Boolean (it is an integer) and, in C#, there is no automatic conversion from 32 to true. Thus, this bug would be caught at compile time, which is a very good thing, and a significant improvement over C++—at the small cost of not allowing implicit conversions from integers to Booleans!

3.5.2.3 Switch statements: an alternative to nested ifs

Nested `if` statements are hard to read, hard to get right, and hard to debug. When you have a complex set of choices to make, the `switch` statement is a more powerful alternative. The logic of a `switch` statement is this: "pick a matching value and act accordingly."

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default: statement]
}
```

As you can see, like an `if` statement, the expression is put in parentheses in the head of the `switch` statement. Each case statement then requires a constant expression; that is, a literal or symbolic constant or an enumeration.

If a case is matched, the statement (or block of statements) associated with that case is executed. This must be followed by a jump statement. Typically, the jump statement is `break`, which transfers execution out of the switch. An alternative is a `goto` statement, typically used to jump into another case, as illustrated in Example 3-9.

Example 3-9. The switch statement

```
using System;


class Values
{
   static void Main(  )
   {
      const int Democrat = 0;
      const int LiberalRepublican = 1;
      const int Republican = 2;
      const int Libertarian = 3;
      const int NewLeft = 4;
      const int Progressive = 5;

      int myChoice = Libertarian;

      switch (myChoice)
      {
         case Democrat:
            Console.WriteLine("You voted Democratic.\n");
            break;
         case LiberalRepublican:  // fall through
            //Console.WriteLine(
                 //"Liberal Republicans vote Republican\n");
         case Republican:
            Console.WriteLine("You voted Republican.\n");
            break;
         case NewLeft:
            Console.Write("NewLeft is now Progressive");
            goto case Progressive;
         case Progressive:
            Console.WriteLine("You voted Progressive.\n");
            break;
         case Libertarian:
            Console.WriteLine("Libertarians are voting Republican");
            goto case Republican;
         default:
            Console.WriteLine("You did not pick a valid choice.\n");
            break;
      }

      Console.WriteLine("Thank you for voting.");

   }
}
```

In this whimsical example, we create constants for various political parties. We then assign one value (`Libertarian`) to the variable `myChoice` and switch on that value. If `myChoice` is equal to `Democrat`, we print out a statement. Notice that this case ends with `break`. `Break` is a jump statement that takes us out of the switch statement and down to the first line after the switch, on which we print "Thank you for voting."

The value `LiberalRepublican` has no statement under it, and it "falls through" to the next statement: `Republican`. If the value is `LiberalRepublican` or `Republican`, the `Republican` statements will execute. You can only "fall through" like this if there is no body within the statement. If you uncomment the `WriteLine` under `LiberalRepublican`, this program will not compile.

C and C++ programmers take note: you cannot fall through to the next case if the `case` statement is not empty. Thus, you can write the following:

```
case 1: // fall through ok
case 2:
```

In this example, `case 1` is empty. You cannot, however, write the following:

```
case 1:
    TakeSomeAction( );
        // fall through not OK
case 2:
```

Here `case 1` has a statement in it, and you cannot fall through. If you want `case 1` to fall through to `case 2`, you must explicitly use `goto`:

```
case 1: TakeSomeAction( );
goto case 2
// explicit fall through
case 2:
```

If you do need a statement but you then want to execute another case, you can use the `goto` statement, as shown in the `NewLeft` case:

```
goto case Progressive;
```

It is not required that the `goto` take you to the case immediately following. In the next instance, the `Libertarian` choice also has a `goto`, but this time it jumps all the way back up to the `Republican` case. Because our value was set to `Libertarian`, this is just what occurs. We print out the `Libertarian` statement, then go to the `Republican` case, print that statement, and then hit the break, taking us out of the switch and down to the final statement. The output for all of this is:

```
Libertarians are voting Republican
You voted Republican.

Thank you for voting.
```

Note the `default` case, excerpted from Example 3-9:

```
default:
    Console.WriteLine(
     "You did not pick a valid choice.\n");
```

If none of the cases matches, the `default` case will be invoked, warning the user of the mistake.

3.5.2.4 Switch on string statements

In the previous example, the switch value was an integral constant. C# offers the ability to switch on a `string`, allowing you to write:

```
case "Libertarian":
```

If the strings match, the `case` statement is entered.

3.5.3 Iteration Statements

C# provides an extensive suite of iteration statements, including `for`, `while` and `do . . . while` loops, as well as `foreach` loops (new to the C family but familiar to VB programmers). In addition, C# supports the `goto`, `break` , `continue`,and `return` jump statements.

3.5.3.1 The goto statement

The `goto` statement is the seed from which all other iteration statements have been germinated. Unfortunately, it is a semolina seed, producer of spaghetti code and endless confusion. Most experienced programmers properly shun the `goto` statement, but in the interest of completeness, here's how you use it:

1. Create a label.
2. `goto` that label.

The label is an identifier followed by a colon. The `goto` command is typically tied to a condition, as illustrated in Example 3-10.

Example 3-10. Using goto

```
using System;
public class Tester
 {

     public static int Main(  )
     {
      int i = 0;
      repeat:            // the label
      Console.WriteLine("i: {0}",i);
      i++;
      if (i < 10)
         goto repeat;  // the dasterdly deed
         return 0;
     }
 }
```

If you were to try to draw the flow of control in a program that makes extensive use of `goto` statements, the resulting morass of intersecting and overlapping lines looks like a plate of spaghetti; hence the term "spaghetti code." It was this phenomenon that led to the creation of alternatives, such as the `while` loop. Many programmers feel that using `goto` in anything other than a trivial example creates confusion and difficult-to-maintain code.

3.5.3.2 The while loop

The semantics of the `while` loop are "while this condition is true, do this work."

The syntax is:

```
while (expression) statement
```

As usual, an expression is any statement that returns a value. `While` statements require an expression that evaluates to a Boolean (`true` /`false`) value, and that statement can, of course, be a block of statements. Example 3-11 updates Example 3-10, using a `while` loop.

Example 3-11. Using a while loop

```
using System;
public class Tester
 {

     public static int Main(  )
     {
      int i = 0;
      while (i < 10)
      {
         Console.WriteLine("i: {0}",i);
         i++;
      }
         return 0;
     }
 }
```

The code in Example 3-11 produces results identical to the code in Example 3-10, but the logic is a bit clearer. The `while` statement is nicely self-contained, and it reads like an English sentence: "`while i` is less than 10, print this message and increment `i`."

Notice that the `while` loop tests the value of `i` before entering the loop. This ensures that the loop will not run if the condition tested is false; thus if `i` is initialized to 11, the loop will never run.

3.5.3.3 The do . . . while loop

There are times when a `while` loop might not serve your purpose. In certain situations, you might want to reverse the semantics from "run while this is true" to the subtly different "do this, while this condition remains true." In other words, take the action, and then, after the action is completed, check the condition. For this you will use the `do...while` loop.

```
do expression while statement
```

An expression is any statement that returns a value. An example of the `do...while` loop is shown in Example 3-12.

Example 3-12. The do...while loop

```
using System;
public class Tester
{

     public static int Main(  )
     {
           int i = 11;
           do
           {
                Console.WriteLine("i: {0}",i);
                i++;
            } while (i < 10);
            return 0;
        }
}
```

Here `i` is initialized to `11` and the `while` test fails, but only after the body of the loop has run once.

3.5.3.4 The for loop

A careful examination of the `while` loop in Example 3-12 reveals a pattern often seen in iterative statements: initialize a variable (`i = 0`), test the variable (`i < 10`), execute a series of statements, and increment the variable (`i++`). The `for` loop allows you to combine all these steps in a single loop statement:

```
for ([initializers]; [expression]; [iterators]) statement
```

The `for` loop is illustrated in Example 3-13.

Example 3-13. The for loop

```
using System;
public class Tester
{

     public static int Main(  )
     {
          for (int i=0;i<100;i++)
          {
               Console.Write("{0} ", i);

               if (i%10 == 0)
               {
                    Console.WriteLine("\t{0}", i);
               }
          }
          return 0;
     }
}
```

```
Output:

0       0
1 2 3 4 5 6 7 8 9 10    10
11 12 13 14 15 16 17 18 19 20   20
21 22 23 24 25 26 27 28 29 30   30
31 32 33 34 35 36 37 38 39 40   40
41 42 43 44 45 46 47 48 49 50   50
51 52 53 54 55 56 57 58 59 60   60
61 62 63 64 65 66 67 68 69 70   70
71 72 73 74 75 76 77 78 79 80   80
81 82 83 84 85 86 87 88 89 90   90
91 92 93 94 95 96 97 98 99
```

This `for` loop makes use of the modulus operator described later in this chapter. The value of `i` is printed until `i` is a multiple of `10`.

```
if (i%10 == 0)
```

A tab is then printed, followed by the value. Thus the tens (20,30,40, etc.) are called out on the right side of the output.

The individual values are printed using `Console.Write`, which is much like `WriteLine` but which does not enter a newline character, allowing the subsequent writes to occur on the same line.

A few quick points to notice: in a `for` loop the condition is tested before the statements are executed. Thus, in the example, `i` is initialized to zero, then `i` is tested to see if it is less than 100. Because i < 100 returns `true`, the statements within the `for` loop are executed. After the execution, `i` is incremented (`i++`).

Note that the variable `i` is scoped to within the `for` loop (that is, the variable `i` is visible only within the `for` loop). Example 3-14 will not compile:

Example 3-14. Scope of variables declared in a for loop

```
using System;
public class Tester
{

      public static int Main(  )
      {
            for (int i=0; i<100; i++)
            {
                  Console.Write("{0} ", i);

                  if ( i%10 == 0 )
                  {
                        Console.WriteLine("\t{0}", i);
                  }
            }
            Console.WriteLine("\n Final value of i: {0}", i);
            return 0;
      }
}
```

The line shown in bold fails, as the variable `i` is not available outside the scope of the `for` loop itself.

Whitespace and Braces

There is much controversy about the use of whitespace in programming. For example, the `for` loop shown in Example 3-14:

```
for (int i=0;i<100;i++)
{
   Console.Write("{0} ", i);

   if (i%10 == 0)
   {

      Console.WriteLine("\t{0}", i);
   }
}
```

could well be written with more space between the operators:

```
         for ( int i = 0; i < 100; i++ )
         {
            Console.Write("{0} ", i);
            if ( i % 10 == 0 )
            {
               Console.WriteLine("\t{0}", i);
            }
         }
```

Because single `for` and `if` statements do not need braces, we can also rewrite the same listing as

```
         for (int i = 0; i < 100; i++)
            Console.Write("{0} ", i);

            if (i % 10 == 0)
               Console.WriteLine("\t{0}", i);
```

Much of this is a matter of personal taste. Although I find whitespace can make code more readable, too much space can cause confusion. In this book, I tend to compress the whitespace to save room on the printed page.

3.5.3.5 The foreach statement

The `foreach` statement is new to the C family of languages; it is used for looping through the elements of an array or a collection. Discussion of this incredibly useful statement is deferred until Chapter 7.

3.5.3.6 The continue and break statements

There are times when you would like to restart a loop without executing the remaining statements in the loop. The `continue` statement causes the loop to return to the top and continue executing.

The obverse side of that coin is the ability to break out of a loop and immediately end all further work within the loop. For this purpose the `break` statement exists.

`Break` and `continue` create multiple exit points and make for hard-to-understand, and thus hard-to-maintain, code. Use them with some care.

Example 3-15 illustrates the mechanics of `continue` and `break`. This code, suggested to me by one of my technical reviewers, Donald Xie, is intended to create a traffic signal processing system. The signals are simulated by entering numerals and uppercase characters from the keyboard, using `Console.ReadLine`, which reads a line of text from the keyboard.

The algorithm is simple: receipt of a "0" (zero) means normal conditions, and no further action is required except to log the event. (In this case, the program simply writes a message to the console; a real application might enter a time-stamped record in a database.) On receipt of an Abort signal (here simulated with an uppercase "A"), the problem is logged and the process is ended. Finally, for any other event, an alarm is raised, perhaps notifying the police. (Note that this sample does not actually notify the police, though it does print out a harrowing message to the console.) If the signal is "X," the alarm is raised but the `while` loop is also terminated.

Example 3-15. Using continue and break

```
using System;
public class Tester
{
   public static int Main(  )
   {
      string signal = "0";      // initialize to neutral
      while (signal != "X")      // X indicates stop
      {
         Console.Write("Enter a signal: ");
         signal = Console.ReadLine(  );

         // do some work here, no matter what signal you
         // receive
         Console.WriteLine("Received: {0}", signal);

         if (signal == "A")
         {
            // faulty - abort signal processing
            // Log the problem and abort.
            Console.WriteLine("Fault! Abort\n");
            break;
         }

         if (signal == "0")
         {
            // normal traffic condition
            // log and continue on
            Console.WriteLine("All is well.\n");
            continue;
         }

         // Problem. Take action and then log the problem
         // and then continue on
         Console.WriteLine("{0} -- raise alarm!\n",
            signal);
      }
      return 0;
   }
}

Output:

Enter a signal: 0
The following signal was received: 0
All is well.

Enter a signal: B
The following signal was received: B
B -- raise alarm!

Enter a signal: A
The following signal was received: A
Faulty processing. Abort

Press any key to continue
```

The point of this exercise is that when the `A` signal is received, the action in the `if` statement is taken and then the program *breaks* out of the loop, without raising the alarm. When the signal is `0` it is also undesirable to raise the alarm, so the program *continues* from the top of the loop.

3.6 Operators

An *operator* is a symbol that causes C# to take an action. The C# primitive types (e.g., `int`) support a number of operators such as assignment, increment, and so forth. Their use is highly intuitive, with the possible exception of the assignment operator (=) and the equality operator (==), which are often confused.

3.6.1 The Assignment Operator (=)

Section 3.3, earlier in this chapter, demonstrates the use of the assignment operator. This symbol causes the operand on the left side of the operator to have its value changed to whatever is on the right side of the operator.

3.6.2 Mathematical Operators

C# uses five mathematical operators, four for standard calculations and a fifth to return the remainder in integer division. The following sections consider the use of these operators.

*3.6.2.1 Simple arithmetical operators (+, -, *, /)*

C# offers operators for simple arithmetic: the addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) operators work as you might expect, with the possible exception of integer division.

When you divide two integers, C# divides like a child in fourth grade: it throws away any fractional remainder. Thus, dividing 17 by 4 will return the value 4 (`17/4 = 4`, with a remainder of `1`). C# provides a special operator, modulus (`%`), described in the next section, to retrieve the remainder.

Note, however, that C# does return fractional answers when you divide floats, doubles, and decimals.

3.6.2.2 The modulus operator (%) to return remainders

To find the remainder in integer division, use the modulus operator (`%`). For example, the statement `17%4` returns `1` (the remainder after integer division).

The modulus operator turns out to be more useful than you might at first imagine. When you perform modulus `n` on a number that is a multiple of *n*, the result is zero. Thus `80 % 10 = 0` because 80 is an even multiple of 10. This fact allows you to set up loops in which you take an action every *n*th time through the loop, by testing a counter to see if `%n` is equal to zero. This strategy comes in handy in the use of the `for` loop, as described earlier in this chapter. The effects of division on integers, floats, doubles, and decimals is illustrated in Example 3-16.

Example 3-16. Division and modulus

```
using System;
class Values
{
   static void Main(  )
   {
      int i1, i2;
      float f1, f2;
      double d1, d2;
      decimal dec1, dec2;
```

```
        i1 = 17;
        i2 = 4;
        f1 = 17f;
        f2 = 4f;
        d1 = 17;
        d2 = 4;
        dec1 = 17;
        dec2 = 4;
        Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
            i1/i2, f1/f2);
        Console.WriteLine("double:\t\t{0}\ndecimal:\t{1}",
            d1/d2, dec1/dec2);
        Console.WriteLine("\nModulus:\t{0}", i1%i2);

    }
}
```

```
Output:

Integer:        4
float:          4.25
double:         4.25
decimal:        4.25

Modulus:        1
```

Now consider this line from Example 3-16:

```
Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
    i1/i2, f1/f2);
```

It begins with a call to `Console.Writeline`, passing in this partial string:

```
"Integer:\t{0}\n
```

This will print the characters `Integer:` followed by a tab `(\t)` followed by the first parameter `({0})` and then followed by a newline character `(\n)`. The next string snippet:

```
float:\t\t{1}\n
```

is very similar. It prints `float:` followed by two tabs (to ensure alignment), the contents of the second parameter `({1})`, and then another newline. Notice the subsequent line, as well:

```
Console.WriteLine("\nModulus:\t{0}", i1%i2);
```

This time the string begins with a newline character, which causes a line to be skipped just before the string `Modulus:` is printed. You can see this effect in the output.

3.6.3 Increment and Decrement Operators

A common requirement is to add a value to a variable, subtract a value from a variable, or otherwise change the mathematical value, and then to assign that new value back to the same variable. You might even want to assign the result to another variable altogether. The following two sections discuss these cases respectively.

3.6.3.1 Calculate and reassign operators

Suppose you want to increment the `mySalary` variable by 5000. You can do this by writing:

```
mySalary = mySalary + 5000;
```

The addition happens before the assignment, and it is perfectly legal to assign the result back to the original variable. Thus, after this operation completes, `mySalary` will have been incremented by 5000. You can perform this kind of assignment with any mathematical operator:

```
mySalary = mySalary * 5000;
mySalary = mySalary - 5000;
```

and so forth.

The need to increment and decrement variables is so common that C# includes special operators for self-assignment. Among these operators are `+=` , `-=`, `*=`, `/=`, and `%=`, which, respectively, combine addition, subtraction, multiplication, division, and modulus, with self-assignment. Thus, you can alternatively write the previous examples as:

```
mySalary += 5000;
mySalary *= 5000;
mySalary -= 5000;
```

The effect of this is to increment `mySalary` by 5000, multiply `mySalary` by 5000, and subtract 5000 from the `mySalary` variable, respectively.

Because incrementing and decrementing by 1 is a very common need, C# (like C and C++ before it) also provides two special operators. To increment by 1 you use the `++` operator, and to decrement by 1 you use the `--` operator.

Thus, if you want to increment the variable `myAge` by 1 you can write:

```
myAge++;
```

3.6.3.2 The prefix and postfix operators

To complicate matters further, you might want to increment a variable and assign the results to a second variable:

```
firstValue = secondValue++;
```

The question arises: do you want to assign before you increment the value or after? In other words, if `secondValue` starts out with the value 10, do you want to end with both `firstValue` and `secondValue` equal to 11, or do you want `firstValue` to be equal to 10 (the original value) and `secondValue` to be equal to 11?

C# (again, like C and C++) offer two flavors of the increment and decrement operators: `prefix` and `postfix`. Thus you can write:

```
firstValue = secondValue++;  // postfix
```

which will assign first, and then increment (`firstValue=10`, `secondValue=11`), or you can write:

```
firstValue = ++secondValue;  //prefix
```

which will increment first, and then assign (`firstValue=11`, `secondValue=11`).

It is important to understand the different effects of `prefix` and `postfix`, as illustrated in Example 3-17.

Example 3-17. Illustrating prefix versus postfix increment

```
using System;
class Values
{
   static void Main(  )
   {
      int valueOne = 10;
      int valueTwo;
      valueTwo = valueOne++;
      Console.WriteLine("After postfix: {0}, {1}", valueOne,
      valueTwo);
      valueOne = 20;
      valueTwo = ++valueOne;
      Console.WriteLine("After prefix: {0}, {1}", valueOne,
      valueTwo);
   }
}
```

Output:

```
After postfix: 11, 10
After prefix: 21, 21
```

3.6.4 Relational Operators

Relational operators are used to compare two values, and then return a Boolean (true or false). The greater-than operator (>), for example, returns true if the value on the left of the operator is greater than the value on the right. Thus, `5 > 2` returns the value `true`, while `2 > 5` returns the value `false`.

The relational operators for C# are shown in Table 3-3. This table assumes two variables: `bigValue` and `smallValue` in which `bigValue` has been assigned the value `100` and `smallValue` the value `50`.

Table 3-3, C# relational operators (assumes bigValue = 100 and smallValue = 50)

Name	Operator	Given this statement:	The expression evaluates to:

Equals	`==`	`bigValue == 100`
		`bigValue = 80`
			`true`
			`false`

Not Equals	`!=`	`bigValue != 100`
		`bigValue != 80`
			`false`
			`true`

Greater than	`>`	`bigValue > smallValue`	`true`

Greater than or equals	`>=`	`bigValue >= smallValue`
		`smallValue >= bigValue`
			`true`
			`false`

Less than	`<`	`bigValue < smallValue`	`false`

Less than or equals	`<=`	`smallValue <= bigValue`
		`bigValue <= smallValue`
			`true`
			`false`

Each of these relational operators acts as you might expect. However, take note of the equals operator (==), which is created by typing two equal signs (=) in a row (i.e., without any space between them); the C# compiler treats the pair as a single operator.

The C# equality operator (==) tests for equality between the objects on either side of the operator. This operator evaluates to a Boolean value (`true` or `false`). Thus, the statement:

```
myX == 5;
```

evaluates to `true` if and only if `myX` is a variable whose value is `5`.

It is not uncommon to confuse the assignment operator (=) with the equals operator (==). The latter has two equal signs, the former only one.

3.6.5 Use of Logical Operators with Conditionals

`If` statements (discussed earlier in this chapter) test whether a condition is true. Often you will want to test whether two conditions are both true, or only one is true, or none is true. C# provides a set of logical operators for this, as shown in Table 3-4. This table assumes two variables, `x` and `y`, in which `x` has the value `5` and `y` the value `7`.

Table 3-4, C# logical operators (assumes x = 5, y = 7)

Name	Operator	Given this statement	The expression evaluates to	Logic
`and`	`&&`	`(x == 3) && (y == 7)`	`false`	Both must be true
`or`	`||`	`(x == 3) || (y == 7)`	`true`	Either or both must be true
`not`	`!`	`! (x == 3)`	`true`	Expression must be false

The `and` operator tests whether two statements are both true. The first line in Table 3-4 includes an example which illustrates the use of the `and` operator:

```
(x == 3) && (y == 7)
```

The entire expression evaluates false because one side (`x == 3`) is false.

With the `or` operator, either or both sides must be true; the expression is false only if both sides are false. So, in the case of the example in Table 3-4:

```
(x == 3) || (y == 7)
```

the entire expression evaluates true because one side (`y==7`) is true.

With a `not` operator, the statement is true if the expression is false, and vice versa. So, in the accompanying example:

```
! (x == 3)
```

the entire expression is true because the tested expression (`x==3`) is false. (The logic is: "it is true that it is not true that x is equal to 3.")

Short-Circuit Evaluation

Consider the following code snippet:

```
int x = 8;
if ((x == 8) || (y == 12))
```

The `if` statement here is a bit complicated. The entire `if` statement is in parentheses, as are all `if` statements in C#. Thus, everything within the outer set of parentheses must evaluate true for the `if` statement to be true.

Within the outer parentheses are two expressions (`x==8`) and (`y==12`) which are separated by an `or` operator (`||`). Because `x` is 8, the first term (`x==8`) evaluates true. There is no need to evaluate the second term (`y==12`). It doesn't matter whether y is 12, the entire expression will be true. Similarly, consider this snippet:

```
int x = 8;
if ((x == 5) && (y == 12))
```

Again, there is no need to evaluate the second term. Because the first term is false, the `and` must fail. (Remember, for an `and` statement to evaluate true, both tested expressions must evaluate true.)

In cases such as these, the C# compiler will short-circuit the evaluation; the second test will never be performed.

3.6.6 Operator Precedence

The compiler must know the order in which to evaluate a series of operators. For example, if I write:

```
myVariable = 5 + 7 * 3;
```

there are three operators for the compiler to evaluate (`=`, `+`, and `*`). It could, for example, operate left to right, which would assign the value `5` to `myVariable`, then add 7 to the 5 (12) and multiply by 3 (36)—but of course then it would throw that 36 away. This is clearly not what is intended.

The rules of precedence tell the compiler which operators to evaluate first. As is the case in algebra, multiplication has higher precedence than addition, so 5+7*3 is equal to 26 rather than 36. Both addition and multiplication have higher precedence than assignment, so the compiler will do the math, and then assign the result (26) to `myVariable` only after the math is completed.

In C#, parentheses are also used to change the order of precedence much as they are in algebra. Thus, you can change the result by writing:

```
myVariable = (5+7) * 3;
```

Grouping the elements of the assignment in this way causes the compiler to add 5+7, multiply the result by 3, and then assign that value (36) to `myVariable`. Table 3-5 summarizes operator precedence in C#.

Table 3-5, Operator precedence

Category	Operators
Primary	`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked`
Unary	+ - ! ~ ++x —x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

In some complex equations you might need to nest your parentheses to ensure the proper order of operations. Assume I want to know how many seconds my family wastes each morning.

It turns out that the adults spend 20 minutes over coffee each morning and 10 minutes reading the newspaper. The children waste 30 minutes dawdling and 10 minutes arguing.

Here's my algorithm:

```
(((minDrinkingCoffee  + minReadingNewspaper )* numAdults ) +
((minDawdling + minArguing) * numChildren)) * secondsPerMinute.
```

Although this works, it is hard to read and hard to get right. It's much easier to use interim variables:

```
wastedByEachAdult = minDrinkingCoffee  +  minReadingNewspaper;
wastedByAllAdults =  wastedByEachAdult * numAdults;
wastedByEachKid =  minDawdling  + minArguing;
wastedByAllKids =  wastedByEachKid * numChildren;
wastedByFamily = wastedByAllAdults + wastedByAllKids;
totalSeconds =  wastedByFamily * 60;
```

The latter example uses many more interim variables, but it is far easier to read, understand, and (most important) debug. As you step through this program in your debugger, you can see the interim values and make sure they are correct.

3.6.7 The Ternary Operator

Although most operators require one term (e.g., `myValue++`) or two terms (e.g., `a+b`), there is one operator that has three—the ternary operator (`?:`).

```
cond-expr ? expr1 : expr2
```

This operator evaluates a *conditional* expression (an expression which returns a value of type `bool`), and then invokes either `expression1` if the value returned from the conditional expression is true, or `expression2` if the value returned is false. The logic is "if this is true, do the first; otherwise do the second." Example 3-18 illustrates.

Example 3-18. The ternary operator

```
using System;
class Values
{
   static void Main(  )
   {
      int valueOne = 10;
      int valueTwo = 20;

        int maxValue = valueOne > valueTwo ?  valueOne : valueTwo;

        Console.WriteLine("ValueOne: {0}, valueTwo: {1}, maxValue: {2}",
            valueOne, valueTwo, maxValue);

   }
}

Output:

ValueOne: 10, valueTwo: 20, maxValue: 20
```

In Example 3-18, the ternary operator is being used to test whether `valueOne` is greater than `valueTwo`. If so, the value of `valueOne` is assigned to the integer variable `maxValue`; otherwise the value of `valueTwo` is assigned to `maxValue`.

3.7 Namespaces

Chapter 2 discusses the reasons for introducing namespaces into the C# language (e.g., avoiding name collisions when using libraries from multiple vendors). In addition to using the namespaces provided by the .NET Framework or other vendors, you are free to create your own. You do this by using the `namespace` keyword, followed by the name you wish to create. Enclose the objects for that namespace within braces, as illustrated in Example 3-19.

Example 3-19. Creating namespaces

```
namespace Programming_C_Sharp
{
   using System;
   public class Tester
   {

      public static int Main(  )
      {
         for (int i=0;i<10;i++)
         {
            Console.WriteLine("i: {0}",i);
         }
         return 0;
      }
   }
}
```

Example 3-19 creates a namespace called `Programming_C_Sharp`, and also specifies a `Tester` class which lives within that namespace. You can alternatively choose to nest your namespaces, as needed, by declaring one within another. You might do so to segment your code, creating objects within a nested namespace whose names are protected from the outer namespace, as illustrated in Example 3-20.

Example 3-20. Nesting namespaces

```
namespace Programming_C_Sharp
{
   namespace Programming_C_Sharp_Test
   {
      using System;
      public class Tester
      {

           public static int Main(  )
           {
               for (int i=0;i<10;i++)
               {
                  Console.WriteLine("i: {0}",i);
               }
              return 0;
           }
       }
   }
}
```

The `Tester` object now declared within the `Programming_C_Sharp_Test` namespace is:

```
Programming_C_Sharp.Programming_C_Sharp_Test.Tester
```

This name would not conflict with another `Tester` object in any other namespace, including the outer namespace `Programming_C_Sharp`.

3.8 Preprocessor Directives

In the examples you've seen so far, you've compiled your entire program whenever you compiled any of it. At times, however, you might want to compile only parts of your program depending on, for example, whether you are debugging or building your production code.

Before your code is compiled, another program called the preprocessor runs and prepares your program for the compiler. The preprocessor examines your code for special preprocessor directives, all of which begin with the pound sign (`#`). These directives allow you to define identifiers and then test for their existence.

3.8.1 Defining Identifiers

`#define DEBUG` defines a preprocessor identifier, `DEBUG`. Although other preprocessor directives can come anywhere in your code, identifiers must be defined before any other code, including `using` statements.

You can test whether `DEBUG` has been defined with the `#if` statement. Thus, you can write:

```
#define DEBUG

//... some normal code - not affected by preprocessor

#if DEBUG
   // code to include if debugging
#else
  // code to include if not debugging
#endif

//... some normal code - not affected by preprocessor
```

When the preprocessor runs, it sees the `#define` statement and records the identifier `DEBUG`. The preprocessor skips over your normal C# code and then finds the `#if - #else - #endif` block.

The `#if` statement tests for the identifier `DEBUG`, which does exist, and so the code between `#if` and `#else` is compiled into your program, but the code between `#else` and `#endif` is *not* compiled. That code does not appear in your assembly at all; it is as if it were left out of your source code.

Had the `#if` statement failed—that is, if you had tested for an identifier which did not exist—the code between `#if` and `#else` would not be compiled, but the code between `#else` and `#endif` would be compiled.

Any code not surrounded by `#if - #endif` is not affected by the preprocessor and is compiled into your program.

3.8.2 Undefining Identifiers

You undefine an identifier with `#undef`. The preprocessor works its way through the code from top to bottom, so the identifier is defined from the `#define` statement until the `#undef` statement, or until the program ends. Thus if you write:

```
#define DEBUG

#if DEBUG
   // this code will be compiled
#endif

#undef DEBUG

#if DEBUG
   // this code will not be compiled
#endif
```

the first `#if` will succeed (`DEBUG` is defined), but the second will fail (`DEBUG` has been undefined).

3.8.3 #if, #elif, #else, and #endif

There is no `switch` statement for the preprocessor, but the `#elif` and `#else` directives provide great flexibility. The `#elif` directive allows the else-if logic of "if DEBUG then action one, else if TEST then action two, else action three":

```
#if DEBUG
   // compile this code if debug is defined
#elif TEST
   // compile this code if debug is not defined
   // but TEST is defined
#else
  // compile this code if neither DEBUG nor TEST
  // is defined
#endif
```

In this example the preprocessor first tests to see if the identifier `DEBUG` is defined. If it is, the code between `#if` and `#elif` will be compiled, and none of the rest of the code until `#endif`, will be compiled.

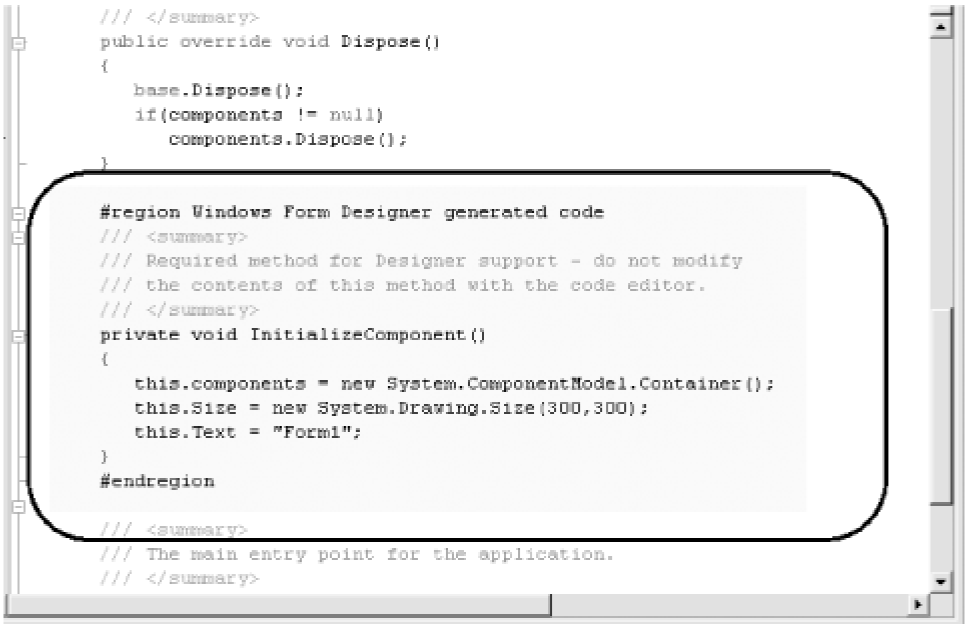
If (and only if) DEBUG is not defined, the preprocessor next checks to see if TEST is defined. Note that the preprocessor will not check for TEST unless DEBUG is not defined. If TEST is defined, the code between the #elif and the #else directives will be compiled. If it turns out that neither DEBUG nor TEST is defined, the code between the #else and the #endif statements will be compiled.

3.8.4 #region

The #region preprocessor directive marks an area of text with a comment. The principal use of this preprocessor directive is to allow tools such as Visual Studio .NET to mark off areas of code and collapse them in the editor with only the region's comment showing.

For example, when you create a Windows application (covered in Chapter 13) Visual Studio .NET creates a region for code generated by the designer. When the region is expanded it looks like Figure 3-1. (Note: I've added the rectangle and highlighting to make it easier to find the region.)

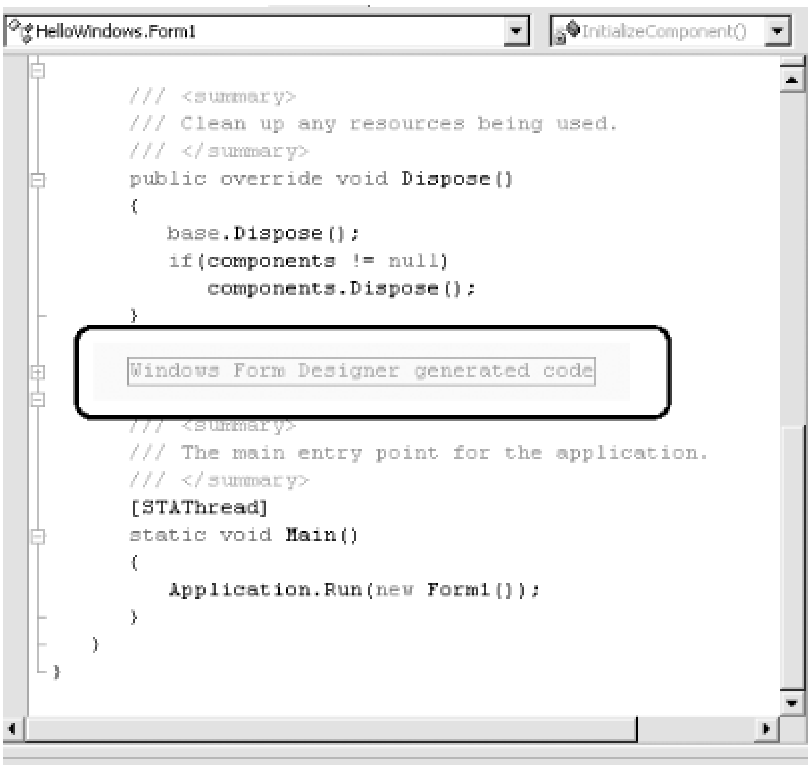
Figure 3-1. Expanding the Visual Studio .NET code region



```
/// </summary>
public override void Dispose()
{
   base.Dispose();
   if(components != null)
      components.Dispose();
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Form1";
}
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
```

You can see the region marked by the #region and #end region preprocessor directives. When the region is collapsed, however, all you see is the region comment (Windows Form Designer generated code), as shown in Figure 3-2.

Figure 3-2. Code region is collapsed



```
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        public override void Dispose()
        {
           base.Dispose();
           if(components != null)
              components.Dispose();
        }
        Windows Form Designer generated code
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
           Application.Run(new Form1());
        }
     }
}
```

Chapter 4. Classes and Objects

Chapter 3 discusses the myriad primitive types built into the C# language, such as `int`, `long`, and `char`. The heart and soul of C#, however, is the ability to create new, complex, programmer-defined types that map cleanly to the objects that make up the problem you are trying to solve.

It is this ability to create new types that characterizes an object-oriented language. You specify new types in C# by declaring and defining classes. You can also define types with interfaces, as you will see in Chapter 8. Instances of a class are called *objects*. Objects are created in memory when your program executes.

The difference between a class and an object is the same as the difference between the concept of a Dog and the particular dog who is sitting at your feet as you read this. You can't play fetch with the definition of a Dog, only with an instance.

A `Dog` class describes what dogs are like: they have weight, height, eye color, hair color, disposition, and so forth. They also have actions they can take, such as eat, walk, bark, and sleep. A particular dog (such as my dog Milo) will have a specific weight (62 pounds), height (22 inches), eye color (black), hair color (yellow), disposition (angelic), and so forth. He is capable of all the actions of any dog (though if you knew him you might imagine that eating is the only method he implements).

The huge advantage of classes in object-oriented programming is that they encapsulate the characteristics and capabilities of an entity in a single, self-contained and self-sustaining *unit of code*. When you want to sort the contents of an instance of a Windows list box control, for example, you tell the list box to sort itself. How it does so is of no concern; *that* it does so is all you need to know. Encapsulation, along with polymorphism and inheritance, is one of three cardinal principles of object-oriented programming.

An old programming joke asks, how many object-oriented programmers does it take to change a light bulb? Answer: none, you just tell the light bulb to change itself. (Alternate answer: none, Microsoft has changed the standard to darkness.)

This chapter explains the C# language features that are used to specify new classes. The elements of a class—its behaviors and properties—are known collectively as its *class members*. This chapter will show how methods are used to define the behaviors of the class, and how the state of the class is maintained in member variables (often called *fields*). In addition, this chapter introduces *properties*, which act like methods to the creator of the class but look like fields to clients of the class.

4.1 Defining Classes

To define a new type or class you first declare it, and then define its methods and fields. You declare a class using the `class` keyword. The complete syntax is as follows:

```
[
attributes

] [
access-modifiers

 ] class identifier  [:base-class ]
{
class-body

 }
```

Attributes are covered in Chapter 18; access modifiers are discussed in the next section. (Typically, your classes will use the keyword `public` as an access modifier.) `The identifier` is the name of the class that you provide. The optional *`base-class`* is discussed in Chapter 5. The member definitions that make up the *`class-body`* are enclosed by open and closed curly braces (`{}`).

> *C++ programmers take note:* a C# class definition does *not* end with a semicolon, though if you add one the program will still compile.

In C#, everything happens within a class. For instance, some of the examples in Chapter 3 make use of a class named `Tester`:

```
public class Tester
{

        public static int Main(  )
        {
         /...
        }
}
```

So far, we've not *instantiated* any instances of that class; that is, we haven't created any `Tester` objects. What is the difference between a class and an instance of that class? To answer that question, start with the distinction between the *type* `int` and a *variable* of type `int`. Thus, while you would write:

```
int myInteger = 5;
```

you would not write:

```
int = 5;
```

You can't assign a value to a type; instead, you assign the value to an object of that type (in this case, a variable of type `int`).

When you declare a new class, you define the properties of all objects of that class, as well as their behaviors. For example, if you are creating a windowing environment, you might want to create screen widgets, more commonly known as controls in Windows programming, to simplify user interaction with your application. One control of interest might be a list box, a control that is very useful for presenting a list of choices to the user and enabling the user to select from the list.

List boxes have a variety of characteristics: height, width, location, and text color, for example. Programmers have also come to expect certain behaviors of list boxes: they can be opened, closed, sorted, and so on.

Object-oriented programming allows you to create a new type, `ListBox,` which encapsulates these characteristics and capabilities. Such a class might have member variables named `height`, `width`, `location,` and `text color`, and member methods named `sort()`, `add()`, `remove( )`, etc.

You can't assign data to the `ListBox` type. Instead you must first create an object of that type, as in the following code snippet:

```
ListBox myListBox;
```

Once you create an instance of `ListBox`, you can assign data to its fields.

Now consider a class to keep track of and display the time of day. The internal state of the class must be able to represent the current year, month, date, hour, minute, and second. You probably would also like the class to display the time in a variety of formats. You might implement such a class by defining a single method and six variables, as shown in Example 4-1.

Example 4-1. Simple Time class

```
using System;

public class Time
{
   // public methods
   public void DisplayCurrentTime(  )
   {
      Console.WriteLine(
         "stub for DisplayCurrentTime");
   }

    // private variables
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;
}

public class Tester
{
   static void Main( )
   {
      Time t = new Time( );
      t.DisplayCurrentTime( );
   }

}
```

The only method declared within the `Time` class definition is the method `DisplayCurrentTime( )`. The body of the method is defined within the class definition itself. Unlike other languages (such as C++), C# does not require that methods be declared before they are defined, nor does the language support placing its declarations into one file and code into another. (C# has no header files.) All C# methods are defined in line as shown in Example 4-1 with `DisplayCurrentTime( )`.

The `DisplayCurrentTime( )` method is defined to return `void`; that is, it will not return a value to a method that invokes it. For now, the body of this method has been "stubbed out."

The `Time` class definition ends with the declaration of a number of member variables: `Year`, `Month`, `Date`, `Hour`, `Minute`, and `Second`.

After the closing brace, a second class, `Tester`, is defined. `Tester` contains our now familiar `Main( )` method. In `Main( )` an instance of `Time` is created and its address is assigned to object `t`. Because `t` is an instance of `Time`, `Main( )` can make use of the `DisplayCurrentTime( )` method available with objects of that type and call it to display the time:

```
t.DisplayCurrentTime( );
```

4.1.1 Access Modifiers

An access modifier determines which class methods—including methods of other classes—can see and use a member variable or method within a class. Table 4-1 summarizes the C# access modifiers.

Table 4-1, Access modifiers

Access Modifier	Restrictions
`public`	No restrictions. Members marked `public` are visible to any method of any class.
`private`	The members in class A which are marked `private` are accessible only to methods of class A.
`protected`	The members in class A which are marked `protected` are accessible to methods of class A and also to methods of classes *derived from* class A.
`internal`	The members in class A which are marked `internal` are accessible to methods of any class in A's assembly.
`protected internal`	The members in class A which are marked `protected internal` are accessible to methods of class A, to methods of classes *derived from* class A, and also to any class in A's assembly. This is effectively `protected` OR `internal` (There is no concept of `protected` AND `internal`.)

It is generally desirable to designate the member variables of a class as `private`. This means that only member methods of that class can access their value. Because `private` is the default accessibility level, you do not need to make it explicit, but I recommend that you do so. Thus, in Example 4-1, the declarations of member variables should have been written as follows:

```
// private variables
```

```
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
```

Class `Tester` and method `DisplayCurrentTime()` are both declared `public` so that any other class can make use of them.

> It is good programming practice to explicitly set the accessibility of all methods and members of your class. Although you can rely on the fact that class members are declared `private` by default, making their access explicit indicates a conscious decision and is self-documenting.

4.1.2 Method Arguments

Methods can take any number of parameters.[1] The parameter list follows the method name and is encased in parentheses, with each parameter preceded by its type. For example, the following declaration defines a method named `MyMethod` which returns `void` (that is, which returns no value at all) and which takes two parameters: an `int` and a button:

[1] The terms "argument" and "parameter" are often used interchangeably, though some programmers insist on differentiating between the argument declaration and the parameters passed in when the method is invoked.

```
void MyMethod (int firstParam, button secondParam)
{
  // ...
}
```

Within the body of the method, the parameters act as local variables, as if you had declared them in the body of the method and initialized them with the values passed in. Example 4-2 illustrates how you pass values into a method, in this case values of type `int` and `float`.

Example 4-2. Passing values into SomeMethod( )

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Here are the parameters received: {0}, {1}",
         firstParam, secondParam);
   }

}

public class Tester
{
   static void Main(  )
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass(  );
      mc.SomeMethod(howManyPeople, pi);
   }
}
```

The method `SomeMethod( )` takes an `int` and a `float` and displays them using `Console.WriteLine( )`. The parameters, which are named `firstParam` and `secondParam`, are treated as local variables within `SomeMethod( )`.

In the calling method `(Main)`, two local variables (`howManyPeople` and `pi)` are created and initialized. These variables are passed as the parameters to `SomeMethod( )`. The compiler maps `howManyPeople` to `firstParam` and `pi` to `secondParam`, based on their relative positions in the parameter list.

4.2 Creating Objects

In Chapter 3, a distinction is drawn between value types and reference types. The primitive C# types (`int`, `char`, etc.) are value types, and are created on the stack. Objects, however, are reference types, and are created on the heap, using the keyword `new`, as in the following:

```
Time t = new Time(  );
```

`t` does not actually contain the value for the `Time` object; it contains the address of that (unnamed) object that is created on the heap. `t` itself is just a reference to that object.

4.2.1 Constructors

In Example 4-1, notice that the statement that creates the `Time` object looks as though it is invoking a method:

```
Time t = new Time(  );
```

In fact, a method *is* invoked whenever you instantiate an object. This method is called a *constructor*, and you must either define one as part of your class definition or let the Common Language Runtime (CLR) provide one on your behalf. The job of a constructor is to create the object specified by a class and to put it into a *valid* state. Before the constructor runs, the object is undifferentiated memory; after the constructor completes, the memory holds a valid instance of the class `type`.

The `Time` class of Example 4-1 does not define a constructor. If a constructor is not declared, the compiler provides one for you. The default constructor creates the object but takes no other action. Member variables are initialized to innocuous values (integers to 0, strings to the empty string, etc.). Table 4-2 lists the default values assigned to primitive types.

Table 4-2, Primitive types and their default values

Type	Default Value
`numeric (int, long`, etc.)	`0`
`bool`	`false`
`char`	`` `\0' (null) ``
`enum`	`0`
`reference`	`null`

Typically, you'll want to define your own constructor and provide it with arguments so that the constructor can set the initial state for your object. In Example 4-1, assume that you want to pass in the current year, month, date, and so forth, so that the object is created with meaningful data.

To define a constructor you declare a method whose name is the same as the class in which it is declared. Constructors have no return type and are typically declared public. If there are arguments to be passed, you define an argument list just as you would for any other method. Example 4-3 declares a constructor for the `Time` class that accepts a single argument, an object of type `DateTime`.

Example 4-3. Declaring a constructor

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime(  )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   // constructor
   public Time(System.DateTime dt)
   {

      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
      Hour = dt.Hour;
      Minute = dt.Minute;
      Second = dt.Second;
   }

    // private member variables
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;

}

public class Tester
{
   static void Main(  )
   {
      System.DateTime currentTime = System.DateTime.Now;
      Time t = new Time(currentTime);
      t.DisplayCurrentTime(  );
   }

}
```

```
 Output:
11/16/2000 16:21:40
```

In this example, the constructor takes a `DateTime` object and initializes all the member variables based on values in that object. When the constructor finishes, the `Time` object exists and the values have been initialized. When `DisplayCurrentTime()` is called in `Main( )`, the values are displayed.

Try commenting out one of the assignments and running the program again. You'll find that the member variable is initialized by the compiler to `0`. Integer member variables are set to `0` if you don't otherwise assign them. Remember, value types (e.g., integers) cannot be *uninitialized*; if you don't tell the constructor what to do, it will try for something innocuous.

In Example 4-3, the `DateTime` object is created in the `Main( )` method of `Tester`. This object, supplied by the `System` library, offers a number of public values—`Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second`—that correspond directly to the private member variables of our `Time` object. In addition, the `DateTime` object offers a static member method, `Now`, which returns a reference to an instance of a `DateTime` object initialized with the current time.

Examine the highlighted line in `Main( )`, where the `DateTime` object is created by calling the static method `Now( )`. `Now( )` creates a `DateTime` object on the heap and returns a reference to it.

That reference is assigned to `currentTime`, which is declared to be a reference to a `DateTime` object. Then `currentTime` is passed as a parameter to the `Time` constructor. The `Time` constructor parameter, `dt`, is also a reference to a `DateTime` object; in fact `dt` now refers to the same `DateTime` object as `currentTime` does. Thus, the `Time` constructor has access to the public member variables of the `DateTime` object that was created in `Tester.Main( )`.

The reason that the `DateTime` object referred to in the `Time` constructor is the same object referred to in `Main( )` is that objects are *reference* types. Thus, when you pass one as a parameter it is passed *by reference*—that is, the pointer is passed and no copy of the object is made.

4.2.2 Initializers

It is possible to initialize the values of member variables in an *initializer*, instead of having to do so in every constructor. You create an initializer by assigning an initial value to a class member:

```
private int Second  = 30;  // initializer
```

Assume that the semantics of our Time object are such that no matter what time is set, the seconds are always initialized to 30. We might rewrite our Time class to use an initializer so that no matter which constructor is called, the value of Second is always initialized, either explicitly by the constructor or implicitly by the initializer, as shown in Example 4-4.

Example 4-4. Using an initializer

```
     public class Time
   {
      // public accessor methods
      public void DisplayCurrentTime(  )
      {
         System.DateTime now = System.DateTime.Now;
            System.Console.WriteLine(
            "\nDebug\t: {0}/{1}/{2} {3}:{4}:{5}",
            now.Month, now.Day , now.Year, now.Hour,
                now.Minute, now.Second);

         System.Console.WriteLine("Time\t: {0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second);
      }


      // constructors
      public Time(System.DateTime dt)
      {

         Year =       dt.Year;
         Month =      dt.Month;
         Date =       dt.Day;
         Hour =      dt.Hour;
         Minute =    dt.Minute;
         Second =    dt.Second;   //explicit assignment
      }

        public Time(int Year, int Month, int Date,
            int Hour, int Minute)
        {
            this.Year =     Year;
            this.Month =    Month;
            this.Date =     Date;
            this.Hour =     Hour;
            this.Minute =   Minute;
        }

       // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second  = 30;  // initializer
   }

   public class Tester
   {
      static void Main(  )
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time(currentTime);
         t.DisplayCurrentTime(  );

         Time t2 = new Time(2000,11,18,11,45);
         t2.DisplayCurrentTime(  );

      }
   }
```

```
 Output:
Debug   : 11/27/2000 7:52:54
Time    : 11/27/2000 7:52:54

Debug   : 11/27/2000 7:52:54
Time    : 11/18/2000 11:45:30
```

If you do not provide a specific initializer, the constructor will initialize each integer member variable to zero (`0`). In the case shown, however, the `Second` member is initialized to `30`:

```
private int Second  = 30;  // initializer
```

If a value is not passed in for `Second`, its value will be set to `30` when `t2` is created:

```
Time t2 = new Time(2000,11,18,11,45);
t2.DisplayCurrentTime(  );
```

However, if a value is assigned to `Second`, as is done in the constructor (which takes a `DateTime` object, shown in bold), that value overrides the initialized value.

The first time through the program we call the constructor that takes a `DateTime` object, and the seconds are initialized to `54`. The second time through we explicitly set the time to `11:45` (not setting the seconds) and the initializer takes over.

If the program did not have an initializer and did not otherwise assign a value to `Second`, the value would be initialized by the compiler to zero.

4.2.4 The this Keyword

The keyword `this` refers to the current instance of an object. The `this` reference (sometimes referred to as a *this pointer*[2]) is a hidden pointer to every nonstatic method of a class. Each method can refer to the other methods and variables of that object by way of the `this` reference.

[2] A pointer is a variable that holds the address of an object in memory. C# does not use pointers with managed objects.

There are three ways in which the `this` reference is typically used. The first way is to qualify instance members otherwise hidden by parameters, as in the following:

```
public void SomeMethod (int hour)
{
   this.hour = hour;
}
```

In this example, `SomeMethod( )` takes a parameter (`Hour`) with the same name as a member variable of the class. The `this` reference is used to resolve the name ambiguity. While `this.Hour` refers to the member variable, `Hour` refers to the parameter.

The argument in favor of this style is that you pick the right variable name and then use it both for the parameter and for the member variable. The counter-argument is that using the same name for both the parameter and the member variable can be confusing.

The second use of the `this` reference is to pass the current object as a parameter to another method. For instance, the following code

```
public void FirstMethod(OtherClass otherObject)
{
   otherObject.SecondMethod(this);
}
```

establishes two classes, one with the method `FirstMethod( )`, and `OtherClass`, with its method `SecondMethod( )`. Inside `FirstMethod`, we'd like to invoke `SecondMethod`, passing in the current object for further processing.

The third use of `this` is with indexers, covered in Chapter 9.

4.3 Using Static Members

The properties and methods of a class can be either *instance members* or *static members*. Instance members are associated with instances of a type, while static members are considered to be part of the class. You access a static member through the name of the class in which it is declared. For example, suppose you have a class named `Button` and have instantiated objects of that class named `btnUpdate` and `btnDelete`. Suppose as well that the `Button` class has a static method `SomeMethod( )`. To access the static method you write:

```
Button.SomeMethod(  );
```

rather than writing:

```
btnUpdate.SomeMethod(  );
```

In C# it is not legal to access a static method or member variable through an instance, and trying to do so will generate a compiler error (C++ programmers, take note).

Some languages distinguish between class methods and other (global) methods that are available outside the context of any class. In C# there are no global methods, only class methods, but you can achieve an analogous result by defining static methods within your class.

Static methods act more or less like global methods, in that you can invoke them without actually having an instance of the object at hand. The advantage of static methods over global, however, is that the name is scoped to the class in which it occurs, and thus you do not clutter up the global namespace with myriad function names. This can help manage highly complex programs, and the name of the class acts very much like a namespace for the static methods within it.

Resist the temptation to create a single class in your program in which you stash all your miscellaneous methods. It is possible but not desirable and undermines the encapsulation of an object-oriented design.

4.3.1 Invoking Static Methods

The `Main( )` method is static. Static methods are said to operate on the class, rather than on an instance of the class. They do not have a `this` reference, as there is no instance to point to.

Static methods cannot directly access nonstatic members. For `Main( )` to call a nonstatic method, it must instantiate an object. Consider Example 4-2, reproduced here for your convenience.

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Here are the parameters received: {0}, {1}",
         firstParam, secondParam);
   }

}

public class Tester
{
   static void Main(  )
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass(  );
      mc.SomeMethod(howManyPeople, pi);
   }

}
```

`SomeMethod( )` is a nonstatic method of `MyClass`. For `Main( )` to access this method, it must first instantiate an object of type `MyClass` and then invoke the method through that object.

4.3.2 Using Static Constructors

If your class declares a static constructor, you will be guaranteed that the static constructor will run before any instance of your class is created.

> You are not able to control exactly when a static constructor will run, but you do know that it will be after the start of your program and before the first instance is created. Because of this you cannot assume (or determine) whether an instance is being created.

For example, you might add the following static constructor to `Time`:

```
static Time(  )
{
     Name = "Time";
}
```

Notice that there is no access modifier (e.g., `public`) before the static constructor. Access modifiers are not allowed on static constructors. In addition, because this is a static member method, you cannot access nonstatic member variables, and so `Name` must be declared a static member variable:

```
private static string Name;
```

The final change is to add a line to `DisplayCurrentTime( )`, as in the following:

```
public void DisplayCurrentTime(  )
{
   System.Console.WriteLine("Name: {0}", Name);
   System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
      Month, Date, Year, Hour, Minute, Second);
}
```

When all these changes are made, the output is:

```
Name: Time
11/20/2000 14:39:8
Name: Time
11/18/2000 11:3:30
Name: Time
11/18/2000 11:3:30
```

Although this code works, it is not necessary to create a static constructor to accomplish this goal. You could, instead, use an initializer:

```
private static string Name = "Time";
```

which accomplishes the same thing. Static constructors are useful, however, for set-up work that cannot be accomplished with an initializer and that needs to be done only once.

For example, assume you have an unmanaged bit of code in a legacy COM dll. You want to provide a class wrapper for this code. You can call load library in your static constructor and initialize the jump table in the static constructor. Handling legacy code and interoperating with unmanaged code is discussed in Chapter 22.

4.3.3 Using Private Constructors

In C# there are no global methods or constants. You might find yourself creating small utility classes that exist only to hold static members. Setting aside whether this is a good design or not, if you create such a class you will not want any instances created. You can prevent any instances from being created by creating a default constructor (one with no parameters) which does nothing, and which is marked `private`. With no public constructors, it will not be possible to create an instance of your class.

4.3.4 Using Static Fields

A common use of static member variables is to keep track of the number of instances that currently exist for your class. Example 4-5 illustrates.

Example 4-5. Using static fields for instance counting

```
using System;

public class Cat
{

     public Cat(  )
     {
          instances++;
     }

     public static void HowManyCats(  )
     {
          Console.WriteLine("{0} cats adopted",
                instances);
     }
     private static int instances = 0;
}

public class Tester
{
     static void Main(  )
     {
           Cat.HowManyCats(  );
           Cat frisky = new Cat(  );
           Cat.HowManyCats(  );
           Cat whiskers = new Cat(  );
           Cat.HowManyCats(  );

     }

}
```

```
Output:

0 cats adopted
1 cats adopted
2 cats adopted
```

The `Cat` class has been stripped to its absolute essentials. A static member variable called `instances` is created and initialized to zero. Note that the static member is considered part of the class, not a member of an instance, and so it cannot be initialized by the compiler on creation of an instance. Thus, an explicit initializer is *required* for static member variables. When additional instances of `Cats` are created (in a constructor) the count is incremented.

Static Methods to Access Static Fields

It is undesirable to make member data `public`. This applies to static member variables as well. One solution is to make the static member `private`, as we've done here with `instances`. We have created a public accessor method, `HowManyCats( )`, to provide access to this private member. Because `HowManyCats( )` is also static, it has access to the static member `instances`.

4.4 Destroying Objects

C# provides garbage collection and thus does not need an explicit destructor. If you do control an unmanaged resource, however, you will need to explicitly free that resource when you are done with it. Implicit control over this resource is provided with a `Finalize( )` method (called a *finalizer*), which will be called by the garbage collector when your object is destroyed.

The finalizer should only release resources that your object holds on to, and should not reference other objects. Note that if you have only managed references you do not need to and should not implement the `Finalize()` method; you want this only for handling unmanaged resources. Because there is some cost to having a finalizer, you ought to implement this only on methods that require it (that is, methods that consume valuable unmanaged resources).

You must never call an object's `Finalize( )` method directly (except that you can call the base class' `Finalize( )` method in your own `Finalize( )`). The garbage collector will call `Finalize( )` for you.

How Finalize Works

The garbage collector maintains a list of objects that have a `Finalize( )` method. This list is updated every time a finalizable object is created or destroyed.

When an object on the garbage collector's finalizable list is first collected, it is placed on a queue with other objects waiting to be finalized. After the `Finalize( )` method executes, the garbage collector then collects the object and updates the queue, as well as its list of finalizable objects.

4.4.1 The C# Destructor

C#'s destructor looks, syntactically, much like a C++ destructor, but it behaves quite differently. You declare a C# destructor with a tilde as follows:

```
~MyClass( ){}
```

In C#, however, this syntax is simply a shortcut for declaring a `Finalize( )` method that chains up to its base class. Thus, writing:

```
~MyClass( )
{
   // do work here
}
```

is identical to writing:

```
MyClass.Finalize( )
{
   // do work here
   base.Finalize( );
}
```

Because of the potential for confusion, it is recommended that you eschew the destructor and write an explicit finalizer if needed.

4.4.2 Finalize Versus Dispose

It is not legal to call a finalizer explicitly. Your `Finalize( )` method will be called by the garbage collector. If you do handle precious unmanaged resources (such as file handles) that you want to close and dispose of as quickly as possible, you ought to implement the `IDisposable` interface. (You will learn more about interfaces in Chapter 8.) The `IDisposable` interface requires its implementers to define one method, named `Dispose( )`, to perform whatever cleanup you consider to be crucial. The availability of `Dispose( )` is a way for your clients to say "don't wait for `Finalize( )` to be called, do it right now."

If you provide a `Dispose( )` method, you should stop the garbage collector from calling `Finalize( )` on your object. To stop the garbage collector, you call the static method `GC.SuppressFinalize()`, passing in the `this` pointer for your object. Your `Finalize( )` method can then call your `Dispose( )` method. Thus, you might write:

```
public void Dispose(  )
{
  // perform clean up

  // tell the GC not to finailze
  GC.SuppressFinalize(this);
}

public override void Finalize(  )
{
  Dispose(  );
  base.Finalize(  );
}
```

4.4.3 Implementing the Close Method

For some objects, you'd rather have your clients call the `Close( )` method. (For example, `Close` makes more sense than `Dispose( )` for file objects.) You can implement this by creating a private `Dispose( )` method and a public `Close( )` method and having your `Close( )` method invoke `Dispose( )`.

4.4.4 The using Statement

Because you cannot be certain that your user will call `Dispose( )` reliably, and because finalization is nondeterministic (i.e., you can't control when the GC will run), C# provides a `using` statement which ensures that `Dispose( )` will be called at the earliest possible time. The idiom is to declare which objects you are using and then to create a scope for these objects with curly braces. When the close brace is reached, the `Dispose( )` method will be called on the object automatically, as illustrated in Example 4-6.

Example 4-6. The using construct

```
using System.Drawing;
class Tester
{
   public static void Main(  )
   {
      using (Font theFont = new Font("Arial", 10.0f))
      {
         // use theFont

      }   // compiler will call Dispose on theFont

      Font anotherFont = new Font("Courier",12.0f);

      using (anotherFont)
      {
         // use anotherFont

      }  // compiler calls Dispose on anotherFont

   }

}
```

In the first part of this example, the `Font` object is created within the `using` statement. When the `using` statement ends, `Dispose( )` is called on the `Font` object.

In the second part of the example, a `Font` object is created outside of the `using` statement. When we decide to use that font, we put it inside the `using` statement and when that statement ends, once again `Dispose( )` is called.

The `using` statement also protects you against unanticipated exceptions. No matter how control leaves the `using` statement, `Dispose( )` is called. It is as if there were an implicit *try-catch-finally* block. (See Section 11.2 in Chapter 11 for details.)

4.5 Passing Parameters

By default, value types are passed into methods by value (see Section 4.1.2 earlier in this chapter). This means that when a value object is passed to a method, a temporary copy of the object is created within that method. Once the method completes, the copy is discarded. Although passing by value is the normal case, there are times when you will want to pass value objects by reference. C# provides the `ref` parameter modifier for passing value objects into a method by reference and the `out` modifier for those cases in which you want to pass in a `ref` variable without first initializing it. C# also supports the `params` modifier which allows a method to accept a variable number of parameters. The `params` keyword is discussed in Chapter 9.

4.5.1 Passing by Reference

Methods can return only a single value (though that value can be a collection of values). Let's return to the `Time` class and add a `GetTime( )` method which returns the hour, minutes, and seconds.

Because we cannot return three values, perhaps we can pass in three parameters, let the method modify the parameters, and examine the result in the calling method. Example 4-7 shows a first attempt at this.

Example 4-7. Returning values in parameters

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime(  )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   public int GetHour(  )
   {
      return Hour;
   }

     public void GetTime(int h, int m, int s)
     {
         h = Hour;
         m = Minute;
         s = Second;
     }

   // constructor
   public Time(System.DateTime dt)
   {
         Year = dt.Year;
         Month = dt.Month;
         Date = dt.Day;
         Hour = dt.Hour;
         Minute = dt.Minute;
         Second = dt.Second;
      }

       // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second;

   }

   public class Tester
   {
      static void Main( )
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time(currentTime);
         t.DisplayCurrentTime( );

            int theHour = 0;
            int theMinute = 0;
            int theSecond = 0;
            t.GetTime(theHour, theMinute, theSecond);
         System.Console.WriteLine("Current time: {0}:{1}:{2}",
                theHour, theMinute, theSecond);

      }

   }
```

```
 Output:
11/17/2000 13:41:18
Current time: 0:0:0
```

Notice that the "Current time" in the output is `0:0:0`. Clearly, this first attempt did not work. The problem is with the parameters. We pass in three integer parameters to `GetTime( )`, and we modify the parameters in `GetTime( )`, but when the values are accessed back in `Main( )`, they are unchanged. This is because integers are value types, and so are passed by value; a copy is made in `GetTime( )`. What we need is to pass these values by reference.

Two small changes are required. First, change the parameters of the `GetTime` method to indicate that the parameters are `ref` (reference) parameters:

```
public void GetTime(ref int h, ref int m, ref int s)
{
     h = Hour;
     m = Minute;
     s = Second;
}
```

Second, modify the call to `GetTime( )` to pass the arguments as references as well:

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
```

If you leave out the second step of marking the arguments with the keyword `ref`, the compiler will complain that the argument cannot be converted from an `int` to a `ref int`.

The results now show the correct time. By declaring these parameters to be `ref` parameters, you instruct the compiler to pass them by reference. Instead of a copy being made, the parameter in `GetTime( )` is a reference to the same variable (`theHour)` that is created in `Main( )`. When you change these values in `GetTime( )`, the change is reflected in `Main( )`.

Keep in mind that ref parameters are references to the actual original value—it is as if you said "here, work on this one." Conversely, value parameters are copies—it is as if you said "here, work on one *just like* this."

4.5.2 Passing Out Parameters with Definite Assignment

C# imposes *definite assignment* , which requires that all variables be assigned a value before they are used. In Example 4-7, if you don't initialize `theHour`, `theMinute`, and `theSecond` before you pass them as parameters to `GetTime( )`, the compiler will complain. Yet the initialization that is done merely sets their values to `0` before they are passed to the method:

```
int theHour = 0;
int theMinute = 0;
int theSecond = 0;
t.GetTime( ref theHour, ref theMinute, ref theSecond);
```

It seems silly to initialize these values because you immediately pass them by reference into `GetTime` where they'll be changed, but if you don't, the following compiler errors are reported:

```
Use of unassigned local variable 'theHour'
Use of unassigned local variable 'theMinute'
Use of unassigned local variable 'theSecond'
```

C# provides the `out` parameter modifier for this situation. The `out` modifier removes the requirement that a reference parameter be initiailzed. The parameters to `GetTime( )`, for example, provide no information to the method; they are simply a mechanism for getting information out of it. Thus, by marking all three as `out` parameters, you eliminate the need to initialize them outside the method. Within the called method the `out` parameters must be assigned a value before the method returns. Here are the altered parameter declarations for `GetTime( )`:

```
public void GetTime(out int h, out int m, out int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
```

and here is the new invocation of the method in `Main( )`:

```
t.GetTime( out theHour, out theMinute, out theSecond);
```

To summarize, value types are passed into methods by value. `Ref` parameters are used to pass value types into a method by reference. This allows you to retrieve their modified value in the calling method. `Out` parameters are used only to return information from a method. Example 4-8 rewrites Example 4-7 to use all three.

Example 4-8. Using in, out, and ref parameters

```
public class Time
  {
     // public accessor methods
     public void DisplayCurrentTime(  )
     {
        System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
           Month, Date, Year, Hour, Minute, Second);
     }

     public int GetHour(  )
     {
        return Hour;
     }

       public void SetTime(int hr, out int min, ref int sec)
       {
           // if the passed in time is >= 30
           // increment the minute and set second to 0
           // otherwise leave both alone
           if (sec >= 30)
           {
               Minute++;
               Second = 0;
           }
           Hour = hr; // set to value passed in

           // pass the minute and second back out
           min = Minute;
           sec = Second;
       }

     // constructor
     public Time(System.DateTime dt)
     {

        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
     }

      // private member variables
     private int Year;
     private int Month;
     private int Date;
     private int Hour;
     private int Minute;
     private int Second;

  }

  public class Tester
  {
     static void Main(  )
     {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
      Hour =      dt.Hour;
      Minute =    dt.Minute;
      Second =    dt.Second;
   }


   // private member variables
   public static int Year;
   public static int Month;
   public static int Date;
   public static int Hour;
   public static int Minute;
   public static int Second;
}

public class Tester
{
   static void Main(  )
   {
      System.Console.WriteLine ("This year: {0}",
             RightNow.Year.ToString(  ));
      RightNow.Year = 2002;
      System.Console.WriteLine ("This year: {0}",
      RightNow.Year.ToString(  ));
   }
}
```

Output:

```
This year: 2000
This year: 2002
```

This works well enough, until someone comes along and changes one of these values. As the example shows, the `RightNow.Year` value can be changed, for example, to `2002`. This is clearly not what we'd like.

We'd like to mark the static values as constant, but that is not possible because we don't initialize them until the static constructor is executed. C# provides the keyword `readonly` for exactly this purpose. If you change the class member variable declarations as follows:

```
public static readonly int Year;
public static readonly int Month;
public static readonly int Date;
public static readonly int Hour;
public static readonly int Minute;
public static readonly int Second;
```

then comment out the reassignment in `Main( )`:

```
// RightNow.Year = 2002; // error!
```

the program will compile and run as intended.

Programming fundamentals :create and use variables and constants. It then goes on to introduce enumerations, strings, identifiers, expressions, and statements.

The second part of the chapter explains and demonstrates the use of branching, using the `if`, `switch`, `while`, `do...while`, `for`, and `foreach` statements. Also discussed are operators, including the assignment, logical, relational, and mathematical operators. This is followed by an introduction to namespaces and a short tutorial on the C# precompiler.

Although C# is principally concerned with the creation and manipulation of objects, it is best to start with the fundamental building blocks: the elements from which objects are created. These include the built-in types that are an intrinsic part of the C# language as well as the syntactic elements of C#.

3.1 Types

C# is a strongly typed language. In a strongly typed language you must declare the type of each object you create (e.g., integers, floats, strings, windows, buttons, etc.) and the compiler will help you prevent bugs by enforcing that only data of the right type is assigned to those objects. The type of an object signals to the compiler the size of that object (e.g., `int` indicates an object of 4 bytes) and its capabilities (e.g., buttons can be drawn, pressed, and so forth).

Like C++ and Java, C# divides types into two sets: *intrinsic* (built-in) types that the language offers and *user-defined* types that the programmer defines.

C# also divides the set of types into two other categories: *value* types and *reference* types.[1] The principal difference between value and reference types is the manner in which their values are stored in memory. A value type holds its actual value in memory allocated on the stack (or it is allocated as part of a larger reference type object). The address of a reference type variable sits on the stack, but the actual object is stored on the heap.

[1] All the intrinsic types are value types except for `Object` (discussed in Chapter 5) and `String` (discussed in Chapter 10). All user-defined types are reference types except for structs (discussed in Chapter 7).

If you have a very large object, putting it on the heap has many advantages. Chapter 4 discusses the various advantages and disadvantages of working with reference types; the current chapter focuses on the intrinsic value types available in C#.

C# also supports C++ style *pointer* types, but these are rarely used, and only when working with unmanaged code. Unmanaged code is code created outside of the .NET platform, such as COM objects. Working with COM objects is discussed in Chapter 22.

3.1.1 Working with Built-in Types

The C# language offers the usual cornucopia of intrinsic (built-in) types one expects in a modern language, each of which maps to an underlying type supported by the .NET Common Language Specification (CLS). Mapping the C# primitive types to the underlying .NET type ensures that objects created in C# can be used interchangeably with objects created in any other language compliant with the .NET CLS, such as VB .NET.

Each type has a specific and unchanging size. Unlike with C++, a C# `int` is always 4 bytes because it maps to an `Int32` in the .NET CLS. Table 3-1 lists the built-in value types offered by C#.

Table 3-1, C# built-in value types

Type	Size (in bytes)	.NET Type	Description
`byte`	1	`Byte`	Unsigned (values 0-255).
`char`	1	`Char`	Unicode characters.
`bool`	1	`Boolean`	`true` or `false`.
`sbyte`	1	`Sbyte`	Signed (values -128 to 127).
`short`	2	`Int16`	Signed (short) (values -32,768 to 32,767).
`ushort`	2	`Uint16`	Unsigned (short) (values 0 to 65,535).
`int`	4	`Int32`	Signed integer values between -2,147,483,647 and 2,147,483,647.
`uint`	4	`Uint32`	Unsigned integer values between 0 and 4,294,967,295.
`float`	4	`Single`	Floating point number. Holds the values from approximately +/-1.5 * 10^{-45} to approximate +/-3.4 * 10^{38} with 7 significant figures.
`double`	8	`Double`	Double-precision floating point; holds the values from approximately +/-5.0 * 10^{-324} to approximate +/-1.7 * 10^{308} with 15-16 significant figures.
`decimal`	8	`Decimal`	Fixed-precision up to 28 digits and the position of the decimal point. This is typically used in financial calculations. Requires the suffix "m" or "M."
`long`	8	`Int64`	Signed integers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`ulong`	8	`Uint64`	Unsigned integers ranging from 0 to 0xffffffffffffffff.

C and C++ programmers take note: Boolean variables can only have the values `true` or `false`. Integer values do not equate to Boolean values in C# and there is no implicit conversion.

In addition to these primitive types, C# has two other value types: `enum` (considered later in this chapter) and `struct` (see Chapter 4). Chapter 4 also discusses other subtleties of value types such as forcing value types to act as reference types through a process known as *boxing*, and that value types do not "inherit."

The Stack and the Heap

A stack is a data structure used to store items on a last-in first-out basis (like a stack of dishes at the buffet line in a restaurant). *The* stack refers to an area of memory supported by the processor, on which the local variables are stored.

In C#, value types (e.g., integers) are allocated on the stack—an area of memory is set aside for their value, and this area is referred to by the name of the variable.

Reference types (e.g., objects) are allocated on the heap. When an object is allocated on the heap its address is returned, and that address is assigned to a reference.

The garbage collector destroys objects on the stack sometime after the stack frame they are declared within ends. Typically a stack frame is defined by a function. Thus, if you declare a local variable within a function (as explained later in this chapter) the object will be marked for garbage collection after the function ends.

Objects on the heap are garbage collected sometime after the final reference to them is destroyed.

3.1.1.1 Choosing a built-in type

Typically you decide which size integer to use (`short`, `int`, or `long`) based on the magnitude of the value you want to store. For example, a `ushort` can only hold values from 0 through 65,535, while a `ulong` can hold values from 0 through 4,294,967,295.

That said, memory is fairly cheap, and programmer time is increasingly expensive; most of the time you'll simply declare your variables to be of type `int`, unless there is a good reason to do otherwise.

The signed types are the numeric types of choice of most programmers unless they have a good reason to use an unsigned value.

Although you might be tempted to use an unsigned `short` to double the positive values of a signed `short` (moving the maximum positive value from 32,767 up to 65,535), it is easier and preferable to use a signed integer (with a maximum value of 2,147,483,647).

It is better to use an unsigned variable when the fact that the value must be positive is an inherent characteristic of the data. For example, if you had a variable to hold a person's age, you would use an unsigned `int` because an age cannot be negative.

`Float`, `double`, and `decimal` offer varying degrees of size and precision. For most small fractional numbers, `float` is fine. Note that the compiler assumes that any number with a decimal point is a double unless you tell it otherwise. To assign a literal `float`, follow the number with the letter `f`. (Assigning values to literals is discussed in detail later in this chapter.)

```
float someFloat = 57f;
```

The `char` type represents a Unicode character. `char` literals can be simple, Unicode, or escape characters enclosed by single quote marks. For example, `A` is a simple character while `\u0041` is a Unicode character. Escape characters are special two-character tokens in which the first character is a backslash. For example, `\t` is a horizontal tab. The common escape characters are shown in Table 3-2.

Table 3-2, Common escape characters

Char	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\0`	Null
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

3.1.1.2 Converting built-in types

Objects of one type can be converted into objects of another type either implicitly or explicitly. Implicit conversions happen automatically; the compiler takes care of it for you. Explicit conversions happen when you "cast" a value to a different type. The semantics of an explicit conversion are "Hey! Compiler! I know what I'm doing." This is sometimes called "hitting it with the big hammer" and can be very useful or very painful, depending on whether your thumb is in the way of the nail.

Implicit conversions happen automatically and are guaranteed not to lose information. For example, you can implicitly cast from a `short int` (2 bytes) to an `int` (4 bytes) implicitly. No matter what value is in the `short`, it will not be lost when converting to an `int`:

```
short x = 5;
int y = x; // implicit conversion
```

If you convert the other way, however, you certainly can lose information. If the value in the `int` is greater than 32,767 it will be truncated in the conversion. The compiler will not perform an implicit conversion from `int` to `short`:

```
short x;
int y = 500;
x = y;  // won't compile
```

You must explicitly convert using the cast operator:

```
short x;
int y = 500;
x = (short) y;  // OK
```

All of the intrinsic types define their own conversion rules. At times it is convenient to define conversion rules for your user-defined types, as discussed in Chapter 5.

3.2 Variables and Constants

A variable is a storage location with a type. In the preceding examples, both *x* and *y* are variables. Variables can have values assigned to them, and that value can be changed programmatically.

WriteLine( )

The .Net Framework provides a useful method for writing output to the screen. The details of this method, `System.Console.WriteLine( )`, will become clearer as we progress through the book, but the fundamentals are straightforward. You call the method as shown in Example 3-3, passing in a string that you want printed to the console (the screen) and, optionally, parameters that will be substituted. In the following example:

```
System.Console.WriteLine("After assignment, myInt: {0}", myInt);
```

the string "`After assignment, myInt:`" is printed as is, followed by the value in the variable `myInt`. The location of the *substitution parameter* `{0}` specifies where the value of the first output variable, `myInt`, will be displayed, in this case at the end of the string. We'll see a great deal more about `Writeline( )` in coming chapters.

You create a variable by declaring its type and then giving it a name. You can initialize the variable when you declare it, and you can assign a new value to that variable at any time, changing the value held in the variable. This is illustrated in Example 3-1.

Example 3-1. Initializing and assigning a value to a variable

```
class Values
{
   static void Main( )
   {
      int myInt = 7;
      System.Console.WriteLine("Initialized, myInt: {0}",
         myInt);
      myInt = 5;
      System.Console.WriteLine("After assignment, myInt: {0}",
         myInt);
   }
}

 Output:
Initialized, myInt: 7
After assignment, myInt: 5
```

Here we initialize the variable `myInt` to the value `7`, display that value, reassign the variable with the value `5`, and display it again.

3.2.1 Definite Assignment

C# requires that variables be initialized before they are used. To test this rule, change the line that initializes `myInt` in Example 3-1 to:

```
int myInt;
```

and save the revised program shown in Example 3-2.

Example 3-2. Using an uninitialized variable

```
class Values
{
   static void Main( )
   {
      int myInt;
      System.Console.WriteLine
      ("Uninitialized, myInt: {0}",myInt);
      myInt = 5;
      System.Console.WriteLine("Assigned, myInt: {0}", myInt);
   }
}
```

When you try to compile this listing, the C# compiler will display the following error message:

```
3.1.cs(6,55): error CS0165: Use of unassigned local
variable 'myInt'
```

It is not legal to use an uninitialized variable in C#. Example 3-2 is not legal.

So, does this mean you must initialize every variable in a program? In fact, no. You don't actually need to initialize a variable, but you must assign a value to it before you attempt to use it. This is called *definite assignment and C# requires it.* Example 3-3 illustrates a correct program.

Example 3-3. Assigning without initializing

```
class Values
{
   static void Main(  )
   {
      int myInt;
      myInt = 7;
      System.Console.WriteLine("Assigned, myInt: {0}", myInt);
      myInt = 5;
      System.Console.WriteLine("Reassigned, myInt: {0}", myInt);
   }
}
```

3.2.2 Constants

A *constant* is a variable whose value cannot be changed. Variables are a powerful tool, but there are times when you want to manipulate a defined value, one whose value you want to ensure remains constant. For example, you might need to work with the Fahrenheit freezing and boiling points of water in a program simulating a chemistry experiment. Your program will be clearer if you name the variables that store these values `FreezingPoint` and `BoilingPoint`, but you do not want to permit their values to be reassigned. How do you prevent reassignment? The answer is to use a constant.

Constants come in three flavors: *literals*, *symbolic constants,* and *enumerations*. In this assignment:

```
x = 32;
```

the value `32` is a literal constant. The value of 32 is always 32. You can't assign a new value to 32; you can't make 32 represent the value `99` no matter how you might try.

Symbolic constants assign a name to a constant value. You declare a symbolic constant using the `const` keyword and the following syntax:

```
const type identifier = value;
```

A constant must be initialized when it is declared, and once initialized it cannot be altered. For example:

```
const int FreezingPoint = 32;
```

In this declaration, 32 is a literal constant and `FreezingPoint` is a symbolic constant of type `int`. Example 3-4 illustrates the use of symbolic constants.

Example 3-4. Using symbolic constants

```
class Values
{
   static void Main(  )
   {
      const int FreezingPoint = 32;   // degrees Farenheit
      const int BoilingPoint = 212;

      System.Console.WriteLine("Freezing point of water: {0}",
            FreezingPoint );
      System.Console.WriteLine("Boiling point of water: {0}",
            BoilingPoint );
      //BoilingPoint = 21;
```

```
    }
}
```

Example 3-4 creates two symbolic integer constants: `FreezingPoint` and `BoilingPoint`. As a matter of style, constant names are written in Pascal notation, but this is certainly not required by the language.

These constants serve the same purpose as always using the *literal* values `32` and `212` for the freezing and boiling points of water in expressions that require them, but because these constants have names they convey far more meaning. Also, if you decide to switch this program to Celsius, you can reinitialize these constants at compile time, to `0` and `100`, respectively; and all the rest of the code ought to continue to work.

To prove to yourself that the constant cannot be reassigned, try uncommenting the last line of the program (shown in bold). When you recompile you should receive this error:

```
error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
```

3.2.3 Enumerations

Enumerations provide a powerful alternative to constants. An enumeration is a distinct value type, consisting of a set of named constants (called the *enumerator list*).

In Example 3-4, you created two related constants:

```
const int FreezingPoint = 32;
const int BoilingPoint = 212;
```

You might wish to add a number of other useful constants as well to this list, such as:

```
const int LightJacketWeather = 60;
const int SwimmingWeather = 72;
const int WickedCold = 0;
```

This process is somewhat cumbersome, and there is no logical connection among these various constants. C# provides the *enumeration* to solve these problems:

```
enum Temperatures
{
   WickedCold = 0,
   FreezingPoint = 32,
   LightJacketWeather = 60,
   SwimmingWeather = 72,
   BoilingPoint = 212,
}
```

Every enumeration has an underlying type, which can be any integral type (`integer`, `short`, `long`, etc.) except for `char`. The technical definition of an enumeration is:

```
[attributes] [modifiers] enum identifier
      [:base-type] {enumerator-list};
```

The optional attributes and modifiers are considered later in this book. For now, let's focus on the rest of this declaration. An enumeration begins with the keyword `enum`, which is generally followed by an identifier, such as:

```
enum Temperatures
```

The base type is the underlying type for the enumeration. If you leave out this optional value (and often you will) it defaults to `integer`, but you are free to use any of the integral types (e.g., `ushort`, `long`) except for `char`. For example, the following fragment declares an enumeration of unsigned integers (`uint`):

```
enum ServingSizes :uint
{
    Small = 1,
    Regular = 2,
    Large = 3
}
```

Notice that an `enum` declaration ends with the enumerator list. The enumerator list contains the constant assignments for the enumeration, each separated by a comma.

Example 3-5 rewrites Example 3-4 to use an enumeration.

Example 3-5. Using enumerations to simplify your code

```
class Values
{

   enum Temperatures
   {
      WickedCold = 0,
      FreezingPoint = 32,
      LightJacketWeather = 60,
      SwimmingWEather = 72,
      BoilingPoint = 212,
   }

   static void Main(  )
   {

      System.Console.WriteLine("Freezing point of water: {0}",
         Temperatures.FreezingPoint );
      System.Console.WriteLine("Boiling point of water: {0}",
         Temperatures.BoilingPoint );

   }
}
```

As you can see, an `enum` must be qualified by its enumtype (e.g., `Temperatures.WickedCold`).

Each constant in an enumeration corresponds to a numerical value, in this case, an integer. If you don't specifically set it otherwise, the enumeration begins at 0 and each subsequent value counts up from the previous.

If you create the following enumeration:

```
enum SomeValues
{
   First,
   Second,
   Third = 20,
   Fourth
}
```

the value of `First` will be `0`, `Second` will be `1`, `Third` will be `20`, and `Fourth` will be 21.

Enums are formal types; therefore an explicit conversion is required to convert between an enum type and an integral type.

C++ programmers take note: C#'s use of enums is subtly different from C++, which restricts assignment to an enum type from an integer but allows an enum to be promoted to an integer for assignment of an enum to an integer.

3.2.4 Strings

It is nearly impossible to write a C# program without creating strings. A string object holds a string of characters.

You declare a string variable using the `string` keyword much as you would create an instance of any object:

```
string myString;
```

A string literal is created by placing double quotes around a string of letters:

```
"Hello World"
```

It is common to initialize a string variable with a string literal:

```
string myString = "Hello World";
```

Strings will be covered in much greater detail in Chapter 10.

3.2.5 Identifiers

Identifiers are names that programmers choose for their types, methods, variables, constants, objects, and so forth. An identifier must begin with a letter or an underscore.

The Microsoft naming conventions suggest using *camel notation* (initial lowercase such as `someName`) for variable names and *Pascal notation* (initial uppercase such as `SomeOtherName`) for method names and most other identifiers.

Microsoft no longer recommends using Hungarian notation (e.g., `iSomeInteger`) or underscores (e.g., `SOME_VALUE`).

Identifiers cannot clash with keywords. Thus, you cannot create a variable named `int` or `class`. In addition, identifiers are case-sensitive, so C# treats `myVariable` and `MyVariable` as two different variable names.

3.3 Expressions

Statements that evaluate to a value are called *expressions*. You may be surprised how many statements do evaluate to a value. For example, an assignment such as:

```
myVariable = 57;
```

is an expression; it evaluates to the value assigned, in this case, `57`.

Note that the preceding statement assigns the value `57` to the variable `myVariable`. The assignment operator (`=`) does not test equality; rather it causes whatever is on the right side (`57`) to be assigned to whatever is on the left side (`myVariable`). All of the C# operators (including assignment and equality) are discussed later in this chapter (see Section 3.6).

Because `myVariable = 57` is an expression that evaluates to `57`, it can be used as part of another assignment operator, such as:

```
mySecondVariable = myVariable = 57;
```

What happens in this statement is that the literal value `57` is assigned to the variable `myVariable`. The value of that assignment (`57`) is then assigned to the second variable, `mySecondVariable`. Thus, the value `57` is assigned to both variables. You can thus initialize any number of variables to the same value with one statement:

```
a = b = c = d = e = 20;
```

3.4 Whitespace

In the C# language, spaces, tabs, and newlines are considered to be " whitespace" (so named because you see only the white of the underlying "page"). Extra whitespace is generally ignored in C# statements. Thus, you can write:

```
myVariable = 5;
```

or:

```
myVariable    =                              5;
```

and the compiler will treat the two statements as identical.

The exception to this rule is that whitespace within strings is not ignored. If you write:

```
Console.WriteLine("Hello World")
```

each space between "Hello" and "World" is treated as another character in the string.

Most of the time the use of whitespace is intuitive. The key is to use whitespace to make the program more readable to the programmer; the compiler is indifferent.

However, there are instances in which the use of whitespace is quite significant. Although the expression:

```
int x = 5;
```

is the same as:

```
int x=5;
```

it is not the same as:

```
intx=5;
```

The compiler knows that the whitespace on either side of the assignment operator is extra, but the whitespace between the type declaration `int` and the variable name `x` is *not* extra, and is required. This is not surprising; the whitespace allows the compiler to parse the keyword `int` rather than some unknown term `intx`. You are free to add as much or as little whitespace between `int` and `x` as you care to, but there must be at least one whitespace character (typically a space or tab).

Visual Basic programmers take note: in C# the end-of-line has no special significance; statements are ended with semicolons, not newline characters. There is no line continuation character because none is needed.

3.5 Statements

In C# a complete program instruction is called a *statement*. Programs consist of sequences of C# statements. Each statement must end with a semicolon (`;`). For example:

```
int x; // a statement
x = 23; // another statement
int y = x; // yet another statement
```

C# statements are evaluated in order. The compiler starts at the beginning of a statement list and makes its way to the bottom. This would be entirely straightforward, and terribly limiting, were it not for branching. There are two types of branches in a C# program: *unconditional branching* and *conditional branching*.

Program flow is also affected by looping and iteration statements, which are signaled by the keywords `for` , `while`, `do`, `in`, and `foreach`. Iteration is discussed later in this chapter. For now, let's consider some of the more basic methods of conditional and unconditional branching.

3.5.1 Unconditional Branching Statements

An unconditional branch is created in one of two ways. The first way is by invoking a method. When the compiler encounters the name of a method it stops execution in the current method and branches to the newly "called" method. When that method returns a value, execution picks up in the original method on the line just below the method call. Example 3-6 illustrates.

Example 3-6. Calling a method

```
using System;
class Functions
{
    static void Main(  )
    {
        Console.WriteLine("In Main! Calling SomeMethod(  )...");
        SomeMethod(  );
        Console.WriteLine("Back in Main(  ).");

    }
    static void SomeMethod(  )
    {
        Console.WriteLine("Greetings from SomeMethod!");
    }
}

Output:

In Main! Calling SomeMethod(  )...
Greetings from SomeMethod!
Back in Main(  ).
```

Program flow begins in `Main( )` and proceeds until `SomeMethod( )` is invoked (invoking a method is sometimes referred to as "calling" the method). At that point program flow branches to the method. When the method completes, program flow resumes at the next line after the call to that method.

The second way to create an unconditional branch is with one of the unconditional branch keywords: `goto`, `break`, `continue`, `return`, or `statementthrow`. Additional information about the first four jump statements is provided in Section 3.5.2.3, Section 3.5.3.1, and Section 3.5.3.6, later in this chapter. The final statement, `throw`, is discussed in Chapter 9.

3.5.2 Conditional Branching Statements

A conditional branch is created by a conditional statement, which is signaled by keywords such as `if`, `else`, or `switch`. A conditional branch occurs only if the condition expression evaluates true.

3.5.2.1 If...else statements

`If...else` statements branch based on a condition. The condition is an expression, tested in the head of the `if` statement. If the condition evaluates true, the statement (or block of statements) in the body of the `if` statement is executed.

`If` statements may contain an optional `else` statement. The `else` statement is executed only if the expression in the head of the `if` statement evaluates false:

```
if (expression)
   statement1
[else
   statement2]
```

This is the kind of description of the `if` statement you are likely to find in your compiler documentation. It shows you that the `if` statement takes an *expression* (a statement that returns a value) in parentheses, and executes `statement1` if the expression evaluates true. Note that `statement1` can actually be a block of statements within braces.

You can also see that the `else` statement is optional, as it is enclosed in square brackets. Although this gives you the syntax of an `if` statement, an illustration will make its use clear. Example 3-7 illustrates.

Example 3-7. If . . . else statements

```
using System;
class Values
{
   static void Main(  )
   {
      int valueOne = 10;
      int valueTwo = 20;

      if ( valueOne > valueTwo )
      {
         Console.WriteLine(
           "ValueOne: {0} larger than ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         Console.WriteLine(
          "ValueTwo: {0} larger than ValueOne: {1}",
                valueTwo,valueOne);
      }

      valueOne = 30; // set valueOne higher

      if ( valueOne > valueTwo )
      {
         valueTwo = valueOne++;
         Console.WriteLine("\nSetting valueTwo to valueOne value, ");
         Console.WriteLine("and incrementing ValueOne.\n");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         valueOne = valueTwo;
         Console.WriteLine("Setting them equal. ");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
   }
}
```

In Example 3-7, the first `if` statement tests whether `valueOne` is greater than `valueTwo`. The relational operators such as greater than (>), less than (<), and equal to (==) are fairly intuitive to use.

The test of whether `valueOne` is greater than `valueTwo` evaluates false (because `valueOne` is 10 and `valueTwo` is 20 and so `valueOne` is *not* greater than `valueTwo`). The `else` statement is invoked, printing the statement:

```
ValueTwo: 20 is larger than ValueOne: 10
```

The second `if` statement evaluates true and all the statements in the `if` block are evaluated, causing two lines to print:

```
ValueOne was larger. Setting valueTwo to old ValueOne value,
and incrementing ValueOne.

ValueOne: 31  ValueTwo: 30
```

Statement Blocks

Anyplace that C# expects a statement, you can substitute a statement block. A *statement block* is a set of statements surrounded by braces.

Thus, where you might write:

```
if (someCondition)
    someStatement;
```

you can instead write:

```
if(someCondition)
{
    statementOne;
    statementTwo;
    statementThree;
}
```

3.5.2.2 Nested if statements

It is possible, and not uncommon, to nest `if` statements to handle complex conditions. For example, suppose you need to write a program to evaluate the temperature, and specifically to return the following types of information:

- If the temperature is 32 degrees or lower, the program should warn you about ice on the road.
- If the temperature is exactly 32 degrees, the program should tell you that there may be ice patches.
- If the temperature is higher than 32 degrees, the program should assure you that there is no ice.

There are many good ways to write this program. Example 3-8 illustrates one approach, using nested `if` statements.

Example 3-8. Nested if statements

```
using System;
class Values
{
   static void Main(  )
   {
      int temp = 32;

      if (temp <= 32)
      {
         Console.WriteLine("Warning! Ice on road!");
         if (temp == 32)
         {
        Console.WriteLine(
          "Temp exactly freezing, beware of water.");
         }
         else
         {
            Console.WriteLine("Watch for black ice! Temp: {0}", temp);
         }
      }
```

```
    }
}
```

The logic of Example 3-8 is that it tests whether the temperature is less than or equal to 32. If so, it prints a warning:

```
if (temp <= 32)
{
   Console.WriteLine("Warning! Ice on road!");
```

The program then checks whether the temp is equal to 32 degrees. If so, it prints one message; if not, the temp must be less than 32 and the program prints the second message. Notice that this second `if` statement is nested within the first `if`, so the logic of the `else` is: "since it has been established that the temp is less than or equal to 32, and it isn't equal to 32, it must be less than 32."

All Operators Are Not Created Equal

A closer examination of the second `if` statement in Example 3-8 reveals a common potential problem. This `if` statement tests whether the temperature is equal to 32:

```
if (temp == 32)
```

In C and C++, there is an inherent danger in this kind of statement. It's not uncommon for novice programmers to use the assignment operator rather than the equals operator, instead creating the statement:

```
if (temp = 32)
```

This mistake would be difficult to notice, and the result would be that `32` was assigned to `temp`, and then `32` would be returned as the value of the assignment statement. Because any nonzero value evaluates to true in C and C#, the `if` statement would return true. The side effect would be that `temp` would be assigned a value of `32` whether or not it originally had that value. This is a common bug that could easily be overlooked—if the developers of C# had not anticipated it!

C# solves this problem by requiring that `if` statements accept only Boolean values. The `32` returned by the assignment is not Boolean (it is an integer) and, in C#, there is no automatic conversion from 32 to true. Thus, this bug would be caught at compile time, which is a very good thing, and a significant improvement over C++—at the small cost of not allowing implicit conversions from integers to Booleans!

3.5.2.3 Switch statements: an alternative to nested ifs

Nested `if` statements are hard to read, hard to get right, and hard to debug. When you have a complex set of choices to make, the `switch` statement is a more powerful alternative. The logic of a `switch` statement is this: "pick a matching value and act accordingly."

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default: statement]
}
```

As you can see, like an `if` statement, the expression is put in parentheses in the head of the `switch` statement. Each case statement then requires a constant expression; that is, a literal or symbolic constant or an enumeration.

If a case is matched, the statement (or block of statements) associated with that case is executed. This must be followed by a jump statement. Typically, the jump statement is `break`, which transfers execution out of the switch. An alternative is a `goto` statement, typically used to jump into another case, as illustrated in Example 3-9.

Example 3-9. The switch statement

```
using System;


class Values
{
   static void Main(  )
   {
      const int Democrat = 0;
      const int LiberalRepublican = 1;
      const int Republican = 2;
      const int Libertarian = 3;
      const int NewLeft = 4;
      const int Progressive = 5;

      int myChoice = Libertarian;

      switch (myChoice)
      {
         case Democrat:
            Console.WriteLine("You voted Democratic.\n");
            break;
         case LiberalRepublican:  // fall through
            //Console.WriteLine(
                 //"Liberal Republicans vote Republican\n");
         case Republican:
            Console.WriteLine("You voted Republican.\n");
            break;
         case NewLeft:
            Console.Write("NewLeft is now Progressive");
            goto case Progressive;
         case Progressive:
            Console.WriteLine("You voted Progressive.\n");
            break;
         case Libertarian:
            Console.WriteLine("Libertarians are voting Republican");
            goto case Republican;
         default:
            Console.WriteLine("You did not pick a valid choice.\n");
            break;
      }

      Console.WriteLine("Thank you for voting.");

   }
}
```

In this whimsical example, we create constants for various political parties. We then assign one value (`Libertarian`) to the variable `myChoice` and switch on that value. If `myChoice` is equal to `Democrat`, we print out a statement. Notice that this case ends with `break`. `Break` is a jump statement that takes us out of the switch statement and down to the first line after the switch, on which we print "Thank you for voting."

The value `LiberalRepublican` has no statement under it, and it "falls through" to the next statement: `Republican`. If the value is `LiberalRepublican` or `Republican`, the `Republican` statements will execute. You can only "fall through" like this if there is no body within the statement. If you uncomment the `WriteLine` under `LiberalRepublican`, this program will not compile.

C and C++ programmers take note: you cannot fall through to the next case if the `case` statement is not empty. Thus, you can write the following:

```
case 1: // fall through ok
case 2:
```

In this example, `case 1` is empty. You cannot, however, write the following:

```
case 1:
    TakeSomeAction( );
        // fall through not OK
case 2:
```

Here `case 1` has a statement in it, and you cannot fall through. If you want `case 1` to fall through to `case 2`, you must explicitly use `goto`:

```
case 1: TakeSomeAction( );
goto case 2
// explicit fall through
case 2:
```

If you do need a statement but you then want to execute another case, you can use the `goto` statement, as shown in the `NewLeft` case:

```
goto case Progressive;
```

It is not required that the `goto` take you to the case immediately following. In the next instance, the `Libertarian` choice also has a `goto`, but this time it jumps all the way back up to the `Republican` case. Because our value was set to `Libertarian`, this is just what occurs. We print out the `Libertarian` statement, then go to the `Republican` case, print that statement, and then hit the break, taking us out of the switch and down to the final statement. The output for all of this is:

```
Libertarians are voting Republican
You voted Republican.

Thank you for voting.
```

Note the `default` case, excerpted from Example 3-9:

```
default:
    Console.WriteLine(
     "You did not pick a valid choice.\n");
```

If none of the cases matches, the `default` case will be invoked, warning the user of the mistake.

3.5.2.4 Switch on string statements

In the previous example, the switch value was an integral constant. C# offers the ability to switch on a `string`, allowing you to write:

```
case "Libertarian":
```

If the strings match, the `case` statement is entered.

3.5.3 Iteration Statements

C# provides an extensive suite of iteration statements, including `for`, `while` and `do . . . while` loops, as well as `foreach` loops (new to the C family but familiar to VB programmers). In addition, C# supports the `goto`, `break` , `continue`,and `return` jump statements.

3.5.3.1 The goto statement

The `goto` statement is the seed from which all other iteration statements have been germinated. Unfortunately, it is a semolina seed, producer of spaghetti code and endless confusion. Most experienced programmers properly shun the `goto` statement, but in the interest of completeness, here's how you use it:

1. Create a label.
2. `goto` that label.

The label is an identifier followed by a colon. The `goto` command is typically tied to a condition, as illustrated in Example 3-10.

Example 3-10. Using goto

```
using System;
public class Tester
 {

     public static int Main(  )
     {
      int i = 0;
      repeat:            // the label
      Console.WriteLine("i: {0}",i);
      i++;
      if (i < 10)
         goto repeat;  // the dasterdly deed
         return 0;
     }
 }
```

If you were to try to draw the flow of control in a program that makes extensive use of `goto` statements, the resulting morass of intersecting and overlapping lines looks like a plate of spaghetti; hence the term "spaghetti code." It was this phenomenon that led to the creation of alternatives, such as the `while` loop. Many programmers feel that using `goto` in anything other than a trivial example creates confusion and difficult-to-maintain code.

3.5.3.2 The while loop

The semantics of the `while` loop are "while this condition is true, do this work."

The syntax is:

```
while (expression) statement
```

As usual, an expression is any statement that returns a value. `While` statements require an expression that evaluates to a Boolean (`true` /`false`) value, and that statement can, of course, be a block of statements. Example 3-11 updates Example 3-10, using a `while` loop.

Example 3-11. Using a while loop

```
using System;
public class Tester
 {

     public static int Main(  )
     {
      int i = 0;
      while (i < 10)
      {
         Console.WriteLine("i: {0}",i);
         i++;
      }
         return 0;
     }
 }
```

The code in Example 3-11 produces results identical to the code in Example 3-10, but the logic is a bit clearer. The `while` statement is nicely self-contained, and it reads like an English sentence: "`while i` is less than 10, print this message and increment `i`."

Notice that the `while` loop tests the value of `i` before entering the loop. This ensures that the loop will not run if the condition tested is false; thus if `i` is initialized to 11, the loop will never run.

3.5.3.3 The do . . . while loop

There are times when a `while` loop might not serve your purpose. In certain situations, you might want to reverse the semantics from "run while this is true" to the subtly different "do this, while this condition remains true." In other words, take the action, and then, after the action is completed, check the condition. For this you will use the `do...while` loop.

```
do expression while statement
```

An expression is any statement that returns a value. An example of the `do...while` loop is shown in Example 3-12.

Example 3-12. The do...while loop

```
using System;
public class Tester
{

     public static int Main(  )
     {
          int i = 11;
          do
          {
                Console.WriteLine("i: {0}",i);
                i++;
            } while (i < 10);
            return 0;
        }
}
```

Here `i` is initialized to `11` and the `while` test fails, but only after the body of the loop has run once.

3.5.3.4 The for loop

A careful examination of the `while` loop in Example 3-12 reveals a pattern often seen in iterative statements: initialize a variable (`i = 0`), test the variable (`i < 10`), execute a series of statements, and increment the variable (`i++`). The `for` loop allows you to combine all these steps in a single loop statement:

```
for ([initializers]; [expression]; [iterators]) statement
```

The `for` loop is illustrated in Example 3-13.

Example 3-13. The for loop

```
using System;
public class Tester
{

     public static int Main(  )
     {
          for (int i=0;i<100;i++)
          {
               Console.Write("{0} ", i);

               if (i%10 == 0)
               {
                    Console.WriteLine("\t{0}", i);
               }
          }
          return 0;
     }
}

Output:

0       0
1 2 3 4 5 6 7 8 9 10    10
11 12 13 14 15 16 17 18 19 20   20
21 22 23 24 25 26 27 28 29 30   30
31 32 33 34 35 36 37 38 39 40   40
41 42 43 44 45 46 47 48 49 50   50
51 52 53 54 55 56 57 58 59 60   60
61 62 63 64 65 66 67 68 69 70   70
71 72 73 74 75 76 77 78 79 80   80
81 82 83 84 85 86 87 88 89 90   90
91 92 93 94 95 96 97 98 99
```

This `for` loop makes use of the modulus operator described later in this chapter. The value of `i` is printed until `i` is a multiple of `10`.

```
if (i%10 == 0)
```

A tab is then printed, followed by the value. Thus the tens (20,30,40, etc.) are called out on the right side of the output.

The individual values are printed using `Console.Write`, which is much like `WriteLine` but which does not enter a newline character, allowing the subsequent writes to occur on the same line.

A few quick points to notice: in a `for` loop the condition is tested before the statements are executed. Thus, in the example, `i` is initialized to zero, then `i` is tested to see if it is less than 100. Because i < 100 returns `true`, the statements within the `for` loop are executed. After the execution, `i` is incremented (`i++`).

Note that the variable `i` is scoped to within the `for` loop (that is, the variable `i` is visible only within the `for` loop). Example 3-14 will not compile:

Example 3-14. Scope of variables declared in a for loop

```
using System;
public class Tester
{

     public static int Main(  )
     {
          for (int i=0; i<100; i++)
          {
               Console.Write("{0} ", i);

               if ( i%10 == 0 )
               {
                    Console.WriteLine("\t{0}", i);
               }
          }
          Console.WriteLine("\n Final value of i: {0}", i);
          return 0;
     }
}
```

The line shown in bold fails, as the variable `i` is not available outside the scope of the `for` loop itself.

Whitespace and Braces

There is much controversy about the use of whitespace in programming. For example, the `for` loop shown in Example 3-14:

```
for (int i=0;i<100;i++)
{
   Console.Write("{0} ", i);

   if (i%10 == 0)
   {

      Console.WriteLine("\t{0}", i);
   }
}
```

could well be written with more space between the operators:

```
         for ( int i = 0; i < 100; i++ )
         {
            Console.Write("{0} ", i);
            if ( i % 10 == 0 )
            {
               Console.WriteLine("\t{0}", i);
            }
         }
```

Because single `for` and `if` statements do not need braces, we can also rewrite the same listing as

```
         for (int i = 0; i < 100; i++)
            Console.Write("{0} ", i);

            if (i % 10 == 0)
               Console.WriteLine("\t{0}", i);
```

Much of this is a matter of personal taste. Although I find whitespace can make code more readable, too much space can cause confusion. In this book, I tend to compress the whitespace to save room on the printed page.

3.5.3.5 The foreach statement

The `foreach` statement is new to the C family of languages; it is used for looping through the elements of an array or a collection. Discussion of this incredibly useful statement is deferred until Chapter 7.

3.5.3.6 The continue and break statements

There are times when you would like to restart a loop without executing the remaining statements in the loop. The `continue` statement causes the loop to return to the top and continue executing.

The obverse side of that coin is the ability to break out of a loop and immediately end all further work within the loop. For this purpose the `break` statement exists.

`Break` and `continue` create multiple exit points and make for hard-to-understand, and thus hard-to-maintain, code. Use them with some care.

Example 3-15 illustrates the mechanics of `continue` and `break`. This code, suggested to me by one of my technical reviewers, Donald Xie, is intended to create a traffic signal processing system. The signals are simulated by entering numerals and uppercase characters from the keyboard, using `Console.ReadLine`, which reads a line of text from the keyboard.

The algorithm is simple: receipt of a "0" (zero) means normal conditions, and no further action is required except to log the event. (In this case, the program simply writes a message to the console; a real application might enter a time-stamped record in a database.) On receipt of an Abort signal (here simulated with an uppercase "A"), the problem is logged and the process is ended. Finally, for any other event, an alarm is raised, perhaps notifying the police. (Note that this sample does not actually notify the police, though it does print out a harrowing message to the console.) If the signal is "X," the alarm is raised but the `while` loop is also terminated.

Example 3-15. Using continue and break

```
using System;
public class Tester
{
   public static int Main(  )
   {
      string signal = "0";      // initialize to neutral
      while (signal != "X")      // X indicates stop
      {
         Console.Write("Enter a signal: ");
         signal = Console.ReadLine(  );

         // do some work here, no matter what signal you
         // receive
         Console.WriteLine("Received: {0}", signal);

         if (signal == "A")
         {
            // faulty - abort signal processing
            // Log the problem and abort.
            Console.WriteLine("Fault! Abort\n");
            break;
         }

         if (signal == "0")
         {
            // normal traffic condition
            // log and continue on
            Console.WriteLine("All is well.\n");
            continue;
         }

         // Problem. Take action and then log the problem
         // and then continue on
         Console.WriteLine("{0} -- raise alarm!\n",
            signal);
      }
      return 0;
   }
}
```

```
Output:

Enter a signal: 0
The following signal was received: 0
All is well.

Enter a signal: B
The following signal was received: B
B -- raise alarm!

Enter a signal: A
The following signal was received: A
Faulty processing. Abort

Press any key to continue
```

The point of this exercise is that when the `A` signal is received, the action in the `if` statement is taken and then the program *breaks* out of the loop, without raising the alarm. When the signal is `0` it is also undesirable to raise the alarm, so the program *continues* from the top of the loop.

3.6 Operators

An *operator* is a symbol that causes C# to take an action. The C# primitive types (e.g., `int`) support a number of operators such as assignment, increment, and so forth. Their use is highly intuitive, with the possible exception of the assignment operator (`=`) and the equality operator (`==`), which are often confused.

3.6.1 The Assignment Operator (=)

Section 3.3, earlier in this chapter, demonstrates the use of the assignment operator. This symbol causes the operand on the left side of the operator to have its value changed to whatever is on the right side of the operator.

3.6.2 Mathematical Operators

C# uses five mathematical operators, four for standard calculations and a fifth to return the remainder in integer division. The following sections consider the use of these operators.

*3.6.2.1 Simple arithmetical operators (+, -, *, /)*

C# offers operators for simple arithmetic: the addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) operators work as you might expect, with the possible exception of integer division.

When you divide two integers, C# divides like a child in fourth grade: it throws away any fractional remainder. Thus, dividing 17 by 4 will return the value 4 (`17/4 = 4`, with a remainder of `1`). C# provides a special operator, modulus (`%`), described in the next section, to retrieve the remainder.

Note, however, that C# does return fractional answers when you divide floats, doubles, and decimals.

3.6.2.2 The modulus operator (%) to return remainders

To find the remainder in integer division, use the modulus operator (`%`). For example, the statement `17%4` returns `1` (the remainder after integer division).

The modulus operator turns out to be more useful than you might at first imagine. When you perform modulus `n` on a number that is a multiple of *n*, the result is zero. Thus `80 % 10 = 0` because 80 is an even multiple of 10. This fact allows you to set up loops in which you take an action every *n*th time through the loop, by testing a counter to see if `%n` is equal to zero. This strategy comes in handy in the use of the `for` loop, as described earlier in this chapter. The effects of division on integers, floats, doubles, and decimals is illustrated in Example 3-16.

Example 3-16. Division and modulus

```
using System;
class Values
{
   static void Main(  )
   {
      int i1, i2;
      float f1, f2;
      double d1, d2;
      decimal dec1, dec2;
      i1 = 17;
      i2 = 4;
      f1 = 17f;
      f2 = 4f;
      d1 = 17;
      d2 = 4;
      dec1 = 17;
      dec2 = 4;
      Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
          i1/i2, f1/f2);
      Console.WriteLine("double:\t\t{0}\ndecimal:\t{1}",
          d1/d2, dec1/dec2);
      Console.WriteLine("\nModulus:\t{0}", i1%i2);

   }
}
```

```
Output:

Integer:        4
float:          4.25
double:         4.25
decimal:        4.25

Modulus:        1
```

Now consider this line from Example 3-16:

```
Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
    i1/i2, f1/f2);
```

It begins with a call to `Console.Writeline`, passing in this partial string:

```
"Integer:\t{0}\n
```

This will print the characters `Integer:` followed by a tab `(\t)` followed by the first parameter `({0})` and then followed by a newline character `(\n)`. The next string snippet:

```
float:\t\t{1}\n
```

is very similar. It prints `float:` followed by two tabs (to ensure alignment), the contents of the second parameter `({1})`, and then another newline. Notice the subsequent line, as well:

```
Console.WriteLine("\nModulus:\t{0}", i1%i2);
```

This time the string begins with a newline character, which causes a line to be skipped just before the string `Modulus:` is printed. You can see this effect in the output.

3.6.3 Increment and Decrement Operators

A common requirement is to add a value to a variable, subtract a value from a variable, or otherwise change the mathematical value, and then to assign that new value back to the same variable. You might even want to assign the result to another variable altogether. The following two sections discuss these cases respectively.

3.6.3.1 Calculate and reassign operators

Suppose you want to increment the `mySalary` variable by 5000. You can do this by writing:

```
mySalary = mySalary + 5000;
```

The addition happens before the assignment, and it is perfectly legal to assign the result back to the original variable. Thus, after this operation completes, `mySalary` will have been incremented by 5000. You can perform this kind of assignment with any mathematical operator:

```
mySalary = mySalary * 5000;
mySalary = mySalary - 5000;
```

and so forth.

The need to increment and decrement variables is so common that C# includes special operators for self-assignment. Among these operators are `+=` , `-=`, `*=`, `/=`, and `%=`, which, respectively, combine addition, subtraction, multiplication, division, and modulus, with self-assignment. Thus, you can alternatively write the previous examples as:

```
mySalary += 5000;
mySalary *= 5000;
mySalary -= 5000;
```

The effect of this is to increment `mySalary` by 5000, multiply `mySalary` by 5000, and subtract 5000 from the `mySalary` variable, respectively.

Because incrementing and decrementing by 1 is a very common need, C# (like C and C++ before it) also provides two special operators. To increment by 1 you use the `++` operator, and to decrement by 1 you use the `--` operator.

Thus, if you want to increment the variable `myAge` by 1 you can write:

```
myAge++;
```

3.6.3.2 The prefix and postfix operators

To complicate matters further, you might want to increment a variable and assign the results to a second variable:

```
firstValue = secondValue++;
```

The question arises: do you want to assign before you increment the value or after? In other words, if `secondValue` starts out with the value 10, do you want to end with both `firstValue` and `secondValue` equal to 11, or do you want `firstValue` to be equal to 10 (the original value) and `secondValue` to be equal to 11?

C# (again, like C and C++) offer two flavors of the increment and decrement operators: `prefix` and `postfix`. Thus you can write:

```
firstValue = secondValue++;  // postfix
```

which will assign first, and then increment (`firstValue=10`, `secondValue=11`), or you can write:

```
firstValue = ++secondValue;  //prefix
```

which will increment first, and then assign (`firstValue=11`, `secondValue=11`).

It is important to understand the different effects of `prefix` and `postfix`, as illustrated in Example 3-17.

Example 3-17. Illustrating prefix versus postfix increment

```
using System;
class Values
{
   static void Main( )
   {
      int valueOne = 10;
      int valueTwo;
      valueTwo = valueOne++;
      Console.WriteLine("After postfix: {0}, {1}", valueOne,
      valueTwo);
      valueOne = 20;
      valueTwo = ++valueOne;
      Console.WriteLine("After prefix: {0}, {1}", valueOne,
      valueTwo);
   }
}
```

Output:

```
After postfix: 11, 10
After prefix: 21, 21
```

3.6.4 Relational Operators

Relational operators are used to compare two values, and then return a Boolean (true or false). The greater-than operator (>), for example, returns true if the value on the left of the operator is greater than the value on the right. Thus, `5 > 2` returns the value `true`, while `2 > 5` returns the value `false`.

The relational operators for C# are shown in Table 3-3. This table assumes two variables: `bigValue` and `smallValue` in which `bigValue` has been assigned the value `100` and `smallValue` the value `50`.

Table 3-3, C# relational operators (assumes bigValue = 100 and smallValue = 50)

Name	Operator	Given this statement:	The expression evaluates to:

Equals	`==`	`bigValue == 100`
		`bigValue = 80`
			`true`
			`false`

Not Equals	`!=`	`bigValue != 100`
		`bigValue != 80`
			`false`
			`true`

Greater than	`>`	`bigValue > smallValue`	`true`

Greater than or equals	`>=`	`bigValue >= smallValue`
		`smallValue >= bigValue`
			`true`
			`false`

Less than	`<`	`bigValue < smallValue`	`false`

Less than or equals	`<=`	`smallValue <= bigValue`
		`bigValue <= smallValue`
			`true`
			`false`

Each of these relational operators acts as you might expect. However, take note of the equals operator (==), which is created by typing two equal signs (=) in a row (i.e., without any space between them); the C# compiler treats the pair as a single operator.

The C# equality operator (==) tests for equality between the objects on either side of the operator. This operator evaluates to a Boolean value (`true` or `false`). Thus, the statement:

```
myX == 5;
```

evaluates to `true` if and only if `myX` is a variable whose value is `5`.

It is not uncommon to confuse the assignment operator (=) with the equals operator (==). The latter has two equal signs, the former only one.

3.6.5 Use of Logical Operators with Conditionals

`If` statements (discussed earlier in this chapter) test whether a condition is true. Often you will want to test whether two conditions are both true, or only one is true, or none is true. C# provides a set of logical operators for this, as shown in Table 3-4. This table assumes two variables, `x` and `y`, in which `x` has the value `5` and `y` the value `7`.

Table 3-4, C# logical operators (assumes x = 5, y = 7)

Name	Operator	Given this statement	The expression evaluates to	Logic
`and`	`&&`	`(x == 3) && (y == 7)`	`false`	Both must be true
`or`	`||`	`(x == 3) || (y == 7)`	`true`	Either or both must be true
`not`	`!`	`! (x == 3)`	`true`	Expression must be false

The `and` operator tests whether two statements are both true. The first line in Table 3-4 includes an example which illustrates the use of the `and` operator:

```
(x == 3) && (y == 7)
```

The entire expression evaluates false because one side (`x == 3`) is false.

With the `or` operator, either or both sides must be true; the expression is false only if both sides are false. So, in the case of the example in Table 3-4:

```
(x == 3) || (y == 7)
```

the entire expression evaluates true because one side (`y==7`) is true.

With a `not` operator, the statement is true if the expression is false, and vice versa. So, in the accompanying example:

```
! (x == 3)
```

the entire expression is true because the tested expression (`x==3`) is false. (The logic is: "it is true that it is not true that x is equal to 3.")

Short-Circuit Evaluation

Consider the following code snippet:

```
int x = 8;
if ((x == 8) || (y == 12))
```

The `if` statement here is a bit complicated. The entire `if` statement is in parentheses, as are all `if` statements in C#. Thus, everything within the outer set of parentheses must evaluate true for the `if` statement to be true.

Within the outer parentheses are two expressions (`x==8`) and (`y==12`) which are separated by an `or` operator (`||`). Because `x` is 8, the first term (`x==8`) evaluates true. There is no need to evaluate the second term (`y==12`). It doesn't matter whether y is 12, the entire expression will be true. Similarly, consider this snippet:

```
int x = 8;
if ((x == 5) && (y == 12))
```

Again, there is no need to evaluate the second term. Because the first term is false, the `and` must fail. (Remember, for an `and` statement to evaluate true, both tested expressions must evaluate true.)

In cases such as these, the C# compiler will short-circuit the evaluation; the second test will never be performed.

3.6.6 Operator Precedence

The compiler must know the order in which to evaluate a series of operators. For example, if I write:

```
myVariable = 5 + 7 * 3;
```

there are three operators for the compiler to evaluate (`=`, `+`, and `*`). It could, for example, operate left to right, which would assign the value `5` to `myVariable`, then add 7 to the 5 (12) and multiply by 3 (36)—but of course then it would throw that 36 away. This is clearly not what is intended.

The rules of precedence tell the compiler which operators to evaluate first. As is the case in algebra, multiplication has higher precedence than addition, so 5+7*3 is equal to 26 rather than 36. Both addition and multiplication have higher precedence than assignment, so the compiler will do the math, and then assign the result (26) to `myVariable` only after the math is completed.

In C#, parentheses are also used to change the order of precedence much as they are in algebra. Thus, you can change the result by writing:

```
myVariable = (5+7) * 3;
```

Grouping the elements of the assignment in this way causes the compiler to add 5+7, multiply the result by 3, and then assign that value (36) to `myVariable`. Table 3-5 summarizes operator precedence in C#.

Table 3-5, Operator precedence

Category	Operators
Primary	`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked`
Unary	+ - ! ~ ++x —x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

In some complex equations you might need to nest your parentheses to ensure the proper order of operations. Assume I want to know how many seconds my family wastes each morning.

It turns out that the adults spend 20 minutes over coffee each morning and 10 minutes reading the newspaper. The children waste 30 minutes dawdling and 10 minutes arguing.

Here's my algorithm:

```
(((minDrinkingCoffee  + minReadingNewspaper )* numAdults ) +
((minDawdling + minArguing) * numChildren)) * secondsPerMinute.
```

Although this works, it is hard to read and hard to get right. It's much easier to use interim variables:

```
wastedByEachAdult = minDrinkingCoffee  +  minReadingNewspaper;
wastedByAllAdults =  wastedByEachAdult * numAdults;
wastedByEachKid =  minDawdling  + minArguing;
wastedByAllKids =  wastedByEachKid * numChildren;
wastedByFamily = wastedByAllAdults + wastedByAllKids;
totalSeconds =  wastedByFamily * 60;
```

The latter example uses many more interim variables, but it is far easier to read, understand, and (most important) debug. As you step through this program in your debugger, you can see the interim values and make sure they are correct.

3.6.7 The Ternary Operator

Although most operators require one term (e.g., `myValue++`) or two terms (e.g., `a+b`), there is one operator that has three—the ternary operator (`?:`).

```
cond-expr ? expr1 : expr2
```

This operator evaluates a *conditional* expression (an expression which returns a value of type `bool`), and then invokes either `expression1` if the value returned from the conditional expression is true, or `expression2` if the value returned is false. The logic is "if this is true, do the first; otherwise do the second." Example 3-18 illustrates.

Example 3-18. The ternary operator

```
using System;
class Values
{
   static void Main(  )
   {
      int valueOne = 10;
      int valueTwo = 20;

        int maxValue = valueOne > valueTwo ?  valueOne : valueTwo;

        Console.WriteLine("ValueOne: {0}, valueTwo: {1}, maxValue: {2}",
            valueOne, valueTwo, maxValue);

   }
}

Output:

ValueOne: 10, valueTwo: 20, maxValue: 20
```

In Example 3-18, the ternary operator is being used to test whether `valueOne` is greater than `valueTwo`. If so, the value of `valueOne` is assigned to the integer variable `maxValue`; otherwise the value of `valueTwo` is assigned to `maxValue`.

3.7 Namespaces

Chapter 2 discusses the reasons for introducing namespaces into the C# language (e.g., avoiding name collisions when using libraries from multiple vendors). In addition to using the namespaces provided by the .NET Framework or other vendors, you are free to create your own. You do this by using the `namespace` keyword, followed by the name you wish to create. Enclose the objects for that namespace within braces, as illustrated in Example 3-19.

Example 3-19. Creating namespaces

```
namespace Programming_C_Sharp
{
   using System;
   public class Tester
   {

      public static int Main(  )
      {
         for (int i=0;i<10;i++)
         {
            Console.WriteLine("i: {0}",i);
         }
         return 0;
      }
   }
}
```

Example 3-19 creates a namespace called `Programming_C_Sharp`, and also specifies a `Tester` class which lives within that namespace. You can alternatively choose to nest your namespaces, as needed, by declaring one within another. You might do so to segment your code, creating objects within a nested namespace whose names are protected from the outer namespace, as illustrated in Example 3-20.

Example 3-20. Nesting namespaces

```
namespace Programming_C_Sharp
{
   namespace Programming_C_Sharp_Test
   {
      using System;
      public class Tester
      {

           public static int Main(  )
           {
               for (int i=0;i<10;i++)
               {
                  Console.WriteLine("i: {0}",i);
               }
              return 0;
           }
       }
   }
}
```

The `Tester` object now declared within the `Programming_C_Sharp_Test` namespace is:

```
Programming_C_Sharp.Programming_C_Sharp_Test.Tester
```

This name would not conflict with another `Tester` object in any other namespace, including the outer namespace `Programming_C_Sharp`.

3.8 Preprocessor Directives

In the examples you've seen so far, you've compiled your entire program whenever you compiled any of it. At times, however, you might want to compile only parts of your program depending on, for example, whether you are debugging or building your production code.

Before your code is compiled, another program called the preprocessor runs and prepares your program for the compiler. The preprocessor examines your code for special preprocessor directives, all of which begin with the pound sign (`#`). These directives allow you to define identifiers and then test for their existence.

3.8.1 Defining Identifiers

`#define DEBUG` defines a preprocessor identifier, `DEBUG`. Although other preprocessor directives can come anywhere in your code, identifiers must be defined before any other code, including `using` statements.

You can test whether `DEBUG` has been defined with the `#if` statement. Thus, you can write:

```
#define DEBUG

//... some normal code - not affected by preprocessor

#if DEBUG
   // code to include if debugging
#else
  // code to include if not debugging
#endif

//... some normal code - not affected by preprocessor
```

When the preprocessor runs, it sees the #define statement and records the identifier DEBUG. The preprocessor skips over your normal C# code and then finds the #if - #else - #endif block.

The #if statement tests for the identifier DEBUG, which does exist, and so the code between #if and #else is compiled into your program, but the code between #else and #endif is *not* compiled. That code does not appear in your assembly at all; it is as if it were left out of your source code.

Had the #if statement failed—that is, if you had tested for an identifier which did not exist—the code between #if and #else would not be compiled, but the code between #else and #endif would be compiled.

Any code not surrounded by #if - #endif is not affected by the preprocessor and is compiled into your program.

3.8.2 Undefining Identifiers

You undefine an identifier with #undef. The preprocessor works its way through the code from top to bottom, so the identifier is defined from the #define statement until the #undef statement, or until the program ends. Thus if you write:

```
#define DEBUG

#if DEBUG
   // this code will be compiled
#endif

#undef DEBUG

#if DEBUG
   // this code will not be compiled
#endif
```

the first #if will succeed (DEBUG is defined), but the second will fail (DEBUG has been undefined).

3.8.3 #if, #elif, #else, and #endif

There is no switch statement for the preprocessor, but the #elif and #else directives provide great flexibility. The #elif directive allows the else-if logic of "if DEBUG then action one, else if TEST then action two, else action three":

```
#if DEBUG
   // compile this code if debug is defined
#elif TEST
   // compile this code if debug is not defined
   // but TEST is defined
#else
  // compile this code if neither DEBUG nor TEST
  // is defined
#endif
```

In this example the preprocessor first tests to see if the identifier DEBUG is defined. If it is, the code between #if and #elif will be compiled, and none of the rest of the code until #endif, will be compiled.

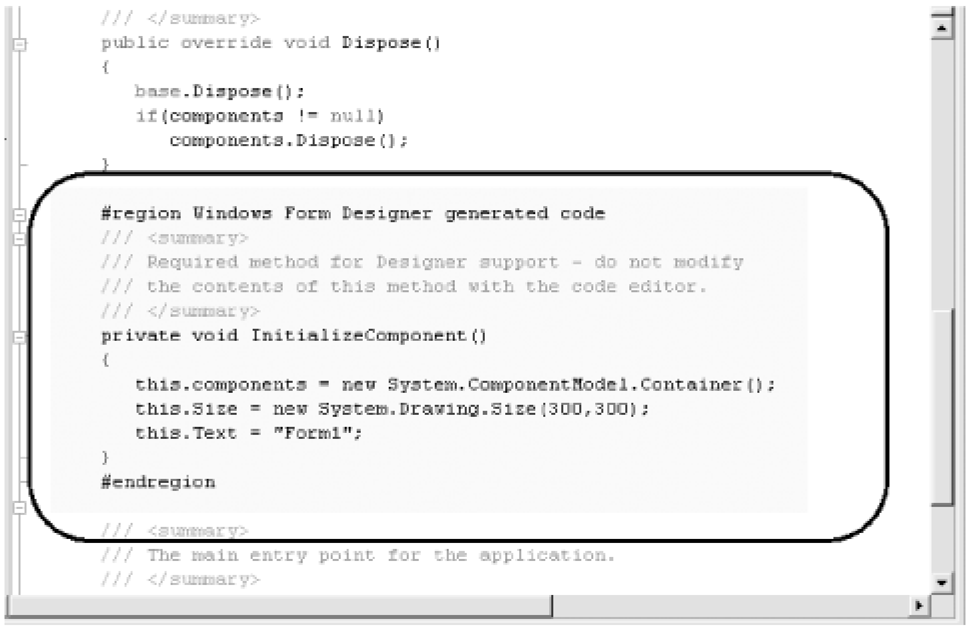
If (and only if) `DEBUG` is not defined, the preprocessor next checks to see if `TEST` is defined. Note that the preprocessor will not check for `TEST` unless `DEBUG` is not defined. If `TEST` is defined, the code between the `#elif` and the `#else` directives will be compiled. If it turns out that neither `DEBUG` nor `TEST` is defined, the code between the `#else` and the `#endif` statements will be compiled.

3.8.4 #region

The `#region` preprocessor directive marks an area of text with a comment. The principal use of this preprocessor directive is to allow tools such as Visual Studio .NET to mark off areas of code and collapse them in the editor with only the region's comment showing.

For example, when you create a Windows application (covered in Chapter 13) Visual Studio .NET creates a region for code generated by the designer. When the region is expanded it looks like Figure 3-1. (Note: I've added the rectangle and highlighting to make it easier to find the region.)

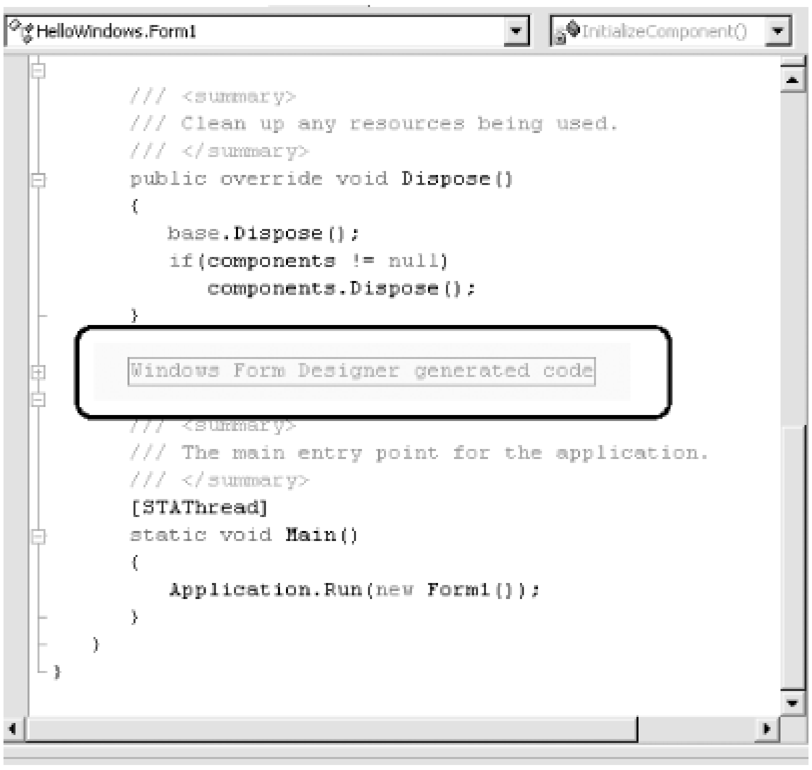
Figure 3-1. Expanding the Visual Studio .NET code region



```
/// </summary>
public override void Dispose()
{
   base.Dispose();
   if(components != null)
      components.Dispose();
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Form1";
}
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
```

You can see the region marked by the `#region` and `#end region` preprocessor directives. When the region is collapsed, however, all you see is the region comment (`Windows Form Designer generated code`), as shown in Figure 3-2.

Figure 3-2. Code region is collapsed


HelloWindows.Form1
InitializeComponent()

```
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        public override void Dispose()
        {
           base.Dispose();
           if(components != null)
              components.Dispose();
        }
        Windows Form Designer generated code
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
           Application.Run(new Form1());
        }
     }
}
```

Chapter 4. Classes and Objects

Chapter 3 discusses the myriad primitive types built into the C# language, such as `int`, `long`, and `char`. The heart and soul of C#, however, is the ability to create new, complex, programmer-defined types that map cleanly to the objects that make up the problem you are trying to solve.

It is this ability to create new types that characterizes an object-oriented language. You specify new types in C# by declaring and defining classes. You can also define types with interfaces, as you will see in Chapter 8. Instances of a class are called *objects*. Objects are created in memory when your program executes.

The difference between a class and an object is the same as the difference between the concept of a Dog and the particular dog who is sitting at your feet as you read this. You can't play fetch with the definition of a Dog, only with an instance.

A `Dog` class describes what dogs are like: they have weight, height, eye color, hair color, disposition, and so forth. They also have actions they can take, such as eat, walk, bark, and sleep. A particular dog (such as my dog Milo) will have a specific weight (62 pounds), height (22 inches), eye color (black), hair color (yellow), disposition (angelic), and so forth. He is capable of all the actions of any dog (though if you knew him you might imagine that eating is the only method he implements).

The huge advantage of classes in object-oriented programming is that they encapsulate the characteristics and capabilities of an entity in a single, self-contained and self-sustaining *unit of code*. When you want to sort the contents of an instance of a Windows list box control, for example, you tell the list box to sort itself. How it does so is of no concern; *that* it does so is all you need to know. Encapsulation, along with polymorphism and inheritance, is one of three cardinal principles of object-oriented programming.

An old programming joke asks, how many object-oriented programmers does it take to change a light bulb? Answer: none, you just tell the light bulb to change itself. (Alternate answer: none, Microsoft has changed the standard to darkness.)

This chapter explains the C# language features that are used to specify new classes. The elements of a class—its behaviors and properties—are known collectively as its *class members*. This chapter will show how methods are used to define the behaviors of the class, and how the state of the class is maintained in member variables (often called *fields*). In addition, this chapter introduces *properties*, which act like methods to the creator of the class but look like fields to clients of the class.

4.1 Defining Classes

To define a new type or class you first declare it, and then define its methods and fields. You declare a class using the `class` keyword. The complete syntax is as follows:

```
[
attributes

] [
access-modifiers

 ] class identifier  [:base-class ]
{
class-body

 }
```

Attributes are covered in Chapter 18; access modifiers are discussed in the next section. (Typically, your classes will use the keyword `public` as an access modifier.) `The identifier` is the name of the class that you provide. The optional *`base-class`* is discussed in Chapter 5. The member definitions that make up the *`class-body`* are enclosed by open and closed curly braces (`{}`).

> *C++ programmers take note:* a C# class definition does *not* end with a semicolon, though if you add one the program will still compile.

In C#, everything happens within a class. For instance, some of the examples in Chapter 3 make use of a class named `Tester`:

```
public class Tester
{

        public static int Main(  )
        {
         /...
        }
}
```

So far, we've not *instantiated* any instances of that class; that is, we haven't created any `Tester` objects. What is the difference between a class and an instance of that class? To answer that question, start with the distinction between the *type* `int` and a *variable* of type `int`. Thus, while you would write:

```
int myInteger = 5;
```

you would not write:

```
int = 5;
```

You can't assign a value to a type; instead, you assign the value to an object of that type (in this case, a variable of type `int`).

When you declare a new class, you define the properties of all objects of that class, as well as their behaviors. For example, if you are creating a windowing environment, you might want to create screen widgets, more commonly known as controls in Windows programming, to simplify user interaction with your application. One control of interest might be a list box, a control that is very useful for presenting a list of choices to the user and enabling the user to select from the list.

List boxes have a variety of characteristics: height, width, location, and text color, for example. Programmers have also come to expect certain behaviors of list boxes: they can be opened, closed, sorted, and so on.

Object-oriented programming allows you to create a new type, `ListBox,` which encapsulates these characteristics and capabilities. Such a class might have member variables named `height`, `width`, `location,` and `text color`, and member methods named `sort()`, `add()`, `remove( )`, etc.

You can't assign data to the `ListBox` type. Instead you must first create an object of that type, as in the following code snippet:

```
ListBox myListBox;
```

Once you create an instance of `ListBox`, you can assign data to its fields.

Now consider a class to keep track of and display the time of day. The internal state of the class must be able to represent the current year, month, date, hour, minute, and second. You probably would also like the class to display the time in a variety of formats. You might implement such a class by defining a single method and six variables, as shown in Example 4-1.

Example 4-1. Simple Time class

```
   using System;

   public class Time
   {
      // public methods
      public void DisplayCurrentTime(  )
      {
         Console.WriteLine(
            "stub for DisplayCurrentTime");
      }

       // private variables
       int Year;
       int Month;
       int Date;
       int Hour;
       int Minute;
       int Second;
   }

   public class Tester
   {
      static void Main( )
      {
         Time t = new Time( );
         t.DisplayCurrentTime( );
      }

   }
```

The only method declared within the `Time` class definition is the method `DisplayCurrentTime( )`. The body of the method is defined within the class definition itself. Unlike other languages (such as C++), C# does not require that methods be declared before they are defined, nor does the language support placing its declarations into one file and code into another. (C# has no header files.) All C# methods are defined in line as shown in Example 4-1 with `DisplayCurrentTime( )`.

The `DisplayCurrentTime( )` method is defined to return `void`; that is, it will not return a value to a method that invokes it. For now, the body of this method has been "stubbed out."

The `Time` class definition ends with the declaration of a number of member variables: `Year`, `Month`, `Date`, `Hour`, `Minute`, and `Second`.

After the closing brace, a second class, `Tester`, is defined. `Tester` contains our now familiar `Main( )` method. In `Main( )` an instance of `Time` is created and its address is assigned to object `t`. Because `t` is an instance of `Time`, `Main( )` can make use of the `DisplayCurrentTime( )` method available with objects of that type and call it to display the time:

```
t.DisplayCurrentTime( );
```

4.1.1 Access Modifiers

An access modifier determines which class methods—including methods of other classes—can see and use a member variable or method within a class. Table 4-1 summarizes the C# access modifiers.

Table 4-1, Access modifiers

Access Modifier	Restrictions
`public`	No restrictions. Members marked `public` are visible to any method of any class.
`private`	The members in class A which are marked `private` are accessible only to methods of class A.
`protected`	The members in class A which are marked `protected` are accessible to methods of class A and also to methods of classes *derived from* class A.
`internal`	The members in class A which are marked `internal` are accessible to methods of any class in A's assembly.
`protected internal`	The members in class A which are marked `protected internal` are accessible to methods of class A, to methods of classes *derived from* class A, and also to any class in A's assembly. This is effectively `protected` OR `internal` (There is no concept of `protected` AND `internal`.)

It is generally desirable to designate the member variables of a class as `private`. This means that only member methods of that class can access their value. Because `private` is the default accessibility level, you do not need to make it explicit, but I recommend that you do so. Thus, in Example 4-1, the declarations of member variables should have been written as follows:

```
// private variables
```

```
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
```

Class `Tester` and method `DisplayCurrentTime()` are both declared `public` so that any other class can make use of them.

> It is good programming practice to explicitly set the accessibility of all methods and members of your class. Although you can rely on the fact that class members are declared `private` by default, making their access explicit indicates a conscious decision and is self-documenting.

4.1.2 Method Arguments

Methods can take any number of parameters.[1] The parameter list follows the method name and is encased in parentheses, with each parameter preceded by its type. For example, the following declaration defines a method named `MyMethod` which returns `void` (that is, which returns no value at all) and which takes two parameters: an `int` and a button:

[1] The terms "argument" and "parameter" are often used interchangeably, though some programmers insist on differentiating between the argument declaration and the parameters passed in when the method is invoked.

```
void MyMethod (int firstParam, button secondParam)
{
  // ...
}
```

Within the body of the method, the parameters act as local variables, as if you had declared them in the body of the method and initialized them with the values passed in. Example 4-2 illustrates how you pass values into a method, in this case values of type `int` and `float`.

Example 4-2. Passing values into SomeMethod( )

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Here are the parameters received: {0}, {1}",
         firstParam, secondParam);
   }

}

public class Tester
{
   static void Main(  )
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass(  );
      mc.SomeMethod(howManyPeople, pi);
   }
}
```

The method `SomeMethod( )` takes an `int` and a `float` and displays them using `Console.WriteLine( )`. The parameters, which are named `firstParam` and `secondParam`, are treated as local variables within `SomeMethod( )`.

In the calling method `(Main)`, two local variables (`howManyPeople` and `pi)` are created and initialized. These variables are passed as the parameters to `SomeMethod( )`. The compiler maps `howManyPeople` to `firstParam` and `pi` to `secondParam`, based on their relative positions in the parameter list.

4.2 Creating Objects

In Chapter 3, a distinction is drawn between value types and reference types. The primitive C# types (`int`, `char`, etc.) are value types, and are created on the stack. Objects, however, are reference types, and are created on the heap, using the keyword `new`, as in the following:

```
Time t = new Time( );
```

`t` does not actually contain the value for the `Time` object; it contains the address of that (unnamed) object that is created on the heap. `t` itself is just a reference to that object.

4.2.1 Constructors

In Example 4-1, notice that the statement that creates the `Time` object looks as though it is invoking a method:

```
Time t = new Time( );
```

In fact, a method *is* invoked whenever you instantiate an object. This method is called a *constructor*, and you must either define one as part of your class definition or let the Common Language Runtime (CLR) provide one on your behalf. The job of a constructor is to create the object specified by a class and to put it into a *valid* state. Before the constructor runs, the object is undifferentiated memory; after the constructor completes, the memory holds a valid instance of the class `type`.

The `Time` class of Example 4-1 does not define a constructor. If a constructor is not declared, the compiler provides one for you. The default constructor creates the object but takes no other action. Member variables are initialized to innocuous values (integers to 0, strings to the empty string, etc.). Table 4-2 lists the default values assigned to primitive types.

Table 4-2, Primitive types and their default values

Type	Default Value
`numeric (int, long`, etc.)	`0`
`bool`	`false`
`char`	`` `\0' (null) ``
`enum`	`0`
`reference`	`null`

Typically, you'll want to define your own constructor and provide it with arguments so that the constructor can set the initial state for your object. In Example 4-1, assume that you want to pass in the current year, month, date, and so forth, so that the object is created with meaningful data.

To define a constructor you declare a method whose name is the same as the class in which it is declared. Constructors have no return type and are typically declared public. If there are arguments to be passed, you define an argument list just as you would for any other method. Example 4-3 declares a constructor for the `Time` class that accepts a single argument, an object of type `DateTime`.

Example 4-3. Declaring a constructor

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime(  )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   // constructor
   public Time(System.DateTime dt)
   {

      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
      Hour = dt.Hour;
      Minute = dt.Minute;
      Second = dt.Second;
   }

    // private member variables
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;

}

public class Tester
{
   static void Main(  )
   {
      System.DateTime currentTime = System.DateTime.Now;
      Time t = new Time(currentTime);
      t.DisplayCurrentTime(  );
   }

}
```

```
 Output:
11/16/2000 16:21:40
```

In this example, the constructor takes a `DateTime` object and initializes all the member variables based on values in that object. When the constructor finishes, the `Time` object exists and the values have been initialized. When `DisplayCurrentTime()` is called in `Main( )`, the values are displayed.

Try commenting out one of the assignments and running the program again. You'll find that the member variable is initialized by the compiler to `0`. Integer member variables are set to `0` if you don't otherwise assign them. Remember, value types (e.g., integers) cannot be *uninitialized*; if you don't tell the constructor what to do, it will try for something innocuous.

In Example 4-3, the `DateTime` object is created in the `Main( )` method of `Tester`. This object, supplied by the `System` library, offers a number of public values—`Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second`—that correspond directly to the private member variables of our `Time` object. In addition, the `DateTime` object offers a static member method, `Now`, which returns a reference to an instance of a `DateTime` object initialized with the current time.

Examine the highlighted line in `Main( )`, where the `DateTime` object is created by calling the static method `Now( )`. `Now( )` creates a `DateTime` object on the heap and returns a reference to it.

That reference is assigned to `currentTime`, which is declared to be a reference to a `DateTime` object. Then `currentTime` is passed as a parameter to the `Time` constructor. The `Time` constructor parameter, `dt`, is also a reference to a `DateTime` object; in fact `dt` now refers to the same `DateTime` object as `currentTime` does. Thus, the `Time` constructor has access to the public member variables of the `DateTime` object that was created in `Tester.Main( )`.

The reason that the `DateTime` object referred to in the `Time` constructor is the same object referred to in `Main( )` is that objects are *reference* types. Thus, when you pass one as a parameter it is passed *by reference*—that is, the pointer is passed and no copy of the object is made.

4.2.2 Initializers

It is possible to initialize the values of member variables in an *initializer*, instead of having to do so in every constructor. You create an initializer by assigning an initial value to a class member:

```
private int Second  = 30;  // initializer
```

Assume that the semantics of our Time object are such that no matter what time is set, the seconds are always initialized to 30. We might rewrite our Time class to use an initializer so that no matter which constructor is called, the value of Second is always initialized, either explicitly by the constructor or implicitly by the initializer, as shown in Example 4-4.

Example 4-4. Using an initializer

```
     public class Time
   {
      // public accessor methods
      public void DisplayCurrentTime(  )
      {
         System.DateTime now = System.DateTime.Now;
            System.Console.WriteLine(
            "\nDebug\t: {0}/{1}/{2} {3}:{4}:{5}",
            now.Month, now.Day , now.Year, now.Hour,
                now.Minute, now.Second);

         System.Console.WriteLine("Time\t: {0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second);
      }


      // constructors
      public Time(System.DateTime dt)
      {

         Year =       dt.Year;
         Month =      dt.Month;
         Date =       dt.Day;
         Hour =      dt.Hour;
         Minute =    dt.Minute;
         Second =    dt.Second;   //explicit assignment
      }

        public Time(int Year, int Month, int Date,
            int Hour, int Minute)
        {
            this.Year =     Year;
            this.Month =    Month;
            this.Date =     Date;
            this.Hour =     Hour;
            this.Minute =   Minute;
        }

       // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second  = 30;  // initializer
   }

   public class Tester
   {
      static void Main(  )
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time(currentTime);
         t.DisplayCurrentTime(  );

         Time t2 = new Time(2000,11,18,11,45);
         t2.DisplayCurrentTime(  );

      }
   }
```

```
 Output:
Debug   : 11/27/2000 7:52:54
Time    : 11/27/2000 7:52:54

Debug   : 11/27/2000 7:52:54
Time    : 11/18/2000 11:45:30
```

If you do not provide a specific initializer, the constructor will initialize each integer member variable to zero (`0`). In the case shown, however, the `Second` member is initialized to `30`:

```
private int Second  = 30;  // initializer
```

If a value is not passed in for `Second`, its value will be set to `30` when `t2` is created:

```
Time t2 = new Time(2000,11,18,11,45);
t2.DisplayCurrentTime(  );
```

However, if a value is assigned to `Second`, as is done in the constructor (which takes a `DateTime` object, shown in bold), that value overrides the initialized value.

The first time through the program we call the constructor that takes a `DateTime` object, and the seconds are initialized to `54`. The second time through we explicitly set the time to `11:45` (not setting the seconds) and the initializer takes over.

If the program did not have an initializer and did not otherwise assign a value to `Second`, the value would be initialized by the compiler to zero.

4.2.4 The this Keyword

The keyword `this` refers to the current instance of an object. The `this` reference (sometimes referred to as a *this pointer*[2]) is a hidden pointer to every nonstatic method of a class. Each method can refer to the other methods and variables of that object by way of the `this` reference.

[2] A pointer is a variable that holds the address of an object in memory. C# does not use pointers with managed objects.

There are three ways in which the `this` reference is typically used. The first way is to qualify instance members otherwise hidden by parameters, as in the following:

```
public void SomeMethod (int hour)
{
   this.hour = hour;
}
```

In this example, `SomeMethod( )` takes a parameter (`Hour`) with the same name as a member variable of the class. The `this` reference is used to resolve the name ambiguity. While `this.Hour` refers to the member variable, `Hour` refers to the parameter.

The argument in favor of this style is that you pick the right variable name and then use it both for the parameter and for the member variable. The counter-argument is that using the same name for both the parameter and the member variable can be confusing.

The second use of the `this` reference is to pass the current object as a parameter to another method. For instance, the following code

```
public void FirstMethod(OtherClass otherObject)
{
   otherObject.SecondMethod(this);
}
```

establishes two classes, one with the method `FirstMethod( )`, and `OtherClass`, with its method `SecondMethod( )`. Inside `FirstMethod,` we'd like to invoke `SecondMethod`, passing in the current object for further processing.

The third use of `this` is with indexers, covered in Chapter 9.

4.3 Using Static Members

The properties and methods of a class can be either *instance members* or *static members*. Instance members are associated with instances of a type, while static members are considered to be part of the class. You access a static member through the name of the class in which it is declared. For example, suppose you have a class named `Button` and have instantiated objects of that class named `btnUpdate` and `btnDelete`. Suppose as well that the `Button` class has a static method `SomeMethod( )`. To access the static method you write:

```
Button.SomeMethod(  );
```

rather than writing:

```
btnUpdate.SomeMethod(  );
```

In C# it is not legal to access a static method or member variable through an instance, and trying to do so will generate a compiler error (C++ programmers, take note).

Some languages distinguish between class methods and other (global) methods that are available outside the context of any class. In C# there are no global methods, only class methods, but you can achieve an analogous result by defining static methods within your class.

Static methods act more or less like global methods, in that you can invoke them without actually having an instance of the object at hand. The advantage of static methods over global, however, is that the name is scoped to the class in which it occurs, and thus you do not clutter up the global namespace with myriad function names. This can help manage highly complex programs, and the name of the class acts very much like a namespace for the static methods within it.

> Resist the temptation to create a single class in your program in which you stash all your miscellaneous methods. It is possible but not desirable and undermines the encapsulation of an object-oriented design.

4.3.1 Invoking Static Methods

The `Main( )` method is static. Static methods are said to operate on the class, rather than on an instance of the class. They do not have a `this` reference, as there is no instance to point to.

Static methods cannot directly access nonstatic members. For `Main( )` to call a nonstatic method, it must instantiate an object. Consider Example 4-2, reproduced here for your convenience.

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Here are the parameters received: {0}, {1}",
         firstParam, secondParam);
   }

}

public class Tester
{
   static void Main(  )
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass(  );
      mc.SomeMethod(howManyPeople, pi);
   }

}
```

`SomeMethod( )` is a nonstatic method of `MyClass`. For `Main( )` to access this method, it must first instantiate an object of type `MyClass` and then invoke the method through that object.

4.3.2 Using Static Constructors

If your class declares a static constructor, you will be guaranteed that the static constructor will run before any instance of your class is created.

> You are not able to control exactly when a static constructor will run, but you do know that it will be after the start of your program and before the first instance is created. Because of this you cannot assume (or determine) whether an instance is being created.

For example, you might add the following static constructor to `Time`:

```
static Time(  )
{
     Name = "Time";
}
```

Notice that there is no access modifier (e.g., `public`) before the static constructor. Access modifiers are not allowed on static constructors. In addition, because this is a static member method, you cannot access nonstatic member variables, and so `Name` must be declared a static member variable:

```
private static string Name;
```

The final change is to add a line to `DisplayCurrentTime( )`, as in the following:

```
public void DisplayCurrentTime(  )
{
   System.Console.WriteLine("Name: {0}", Name);
   System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
      Month, Date, Year, Hour, Minute, Second);
}
```

When all these changes are made, the output is:

```
Name: Time
11/20/2000 14:39:8
Name: Time
11/18/2000 11:3:30
Name: Time
11/18/2000 11:3:30
```

Although this code works, it is not necessary to create a static constructor to accomplish this goal. You could, instead, use an initializer:

```
private static string Name = "Time";
```

which accomplishes the same thing. Static constructors are useful, however, for set-up work that cannot be accomplished with an initializer and that needs to be done only once.

For example, assume you have an unmanaged bit of code in a legacy COM dll. You want to provide a class wrapper for this code. You can call load library in your static constructor and initialize the jump table in the static constructor. Handling legacy code and interoperating with unmanaged code is discussed in Chapter 22.

4.3.3 Using Private Constructors

In C# there are no global methods or constants. You might find yourself creating small utility classes that exist only to hold static members. Setting aside whether this is a good design or not, if you create such a class you will not want any instances created. You can prevent any instances from being created by creating a default constructor (one with no parameters) which does nothing, and which is marked `private`. With no public constructors, it will not be possible to create an instance of your class.

4.3.4 Using Static Fields

A common use of static member variables is to keep track of the number of instances that currently exist for your class. Example 4-5 illustrates.

Example 4-5. Using static fields for instance counting

```
using System;

public class Cat
{

     public Cat(  )
     {
          instances++;
     }

     public static void HowManyCats(  )
     {
          Console.WriteLine("{0} cats adopted",
                instances);
     }
     private static int instances = 0;
}

public class Tester
{
     static void Main(  )
     {
           Cat.HowManyCats(  );
           Cat frisky = new Cat(  );
           Cat.HowManyCats(  );
           Cat whiskers = new Cat(  );
           Cat.HowManyCats(  );

     }

}
```

```
Output:

0 cats adopted
1 cats adopted
2 cats adopted
```

The `Cat` class has been stripped to its absolute essentials. A static member variable called `instances` is created and initialized to zero. Note that the static member is considered part of the class, not a member of an instance, and so it cannot be initialized by the compiler on creation of an instance. Thus, an explicit initializer is *required* for static member variables. When additional instances of `Cats` are created (in a constructor) the count is incremented.

Static Methods to Access Static Fields

It is undesirable to make member data `public`. This applies to static member variables as well. One solution is to make the static member `private`, as we've done here with `instances`. We have created a public accessor method, `HowManyCats( )`, to provide access to this private member. Because `HowManyCats( )` is also static, it has access to the static member `instances`.

4.4 Destroying Objects

C# provides garbage collection and thus does not need an explicit destructor. If you do control an unmanaged resource, however, you will need to explicitly free that resource when you are done with it. Implicit control over this resource is provided with a `Finalize( )` method (called a *finalizer*), which will be called by the garbage collector when your object is destroyed.

The finalizer should only release resources that your object holds on to, and should not reference other objects. Note that if you have only managed references you do not need to and should not implement the `Finalize()` method; you want this only for handling unmanaged resources. Because there is some cost to having a finalizer, you ought to implement this only on methods that require it (that is, methods that consume valuable unmanaged resources).

You must never call an object's `Finalize( )` method directly (except that you can call the base class' `Finalize( )` method in your own `Finalize( )`). The garbage collector will call `Finalize( )` for you.

How Finalize Works

The garbage collector maintains a list of objects that have a `Finalize( )` method. This list is updated every time a finalizable object is created or destroyed.

When an object on the garbage collector's finalizable list is first collected, it is placed on a queue with other objects waiting to be finalized. After the `Finalize( )` method executes, the garbage collector then collects the object and updates the queue, as well as its list of finalizable objects.

4.4.1 The C# Destructor

C#'s destructor looks, syntactically, much like a C++ destructor, but it behaves quite differently. You declare a C# destructor with a tilde as follows:

```
~MyClass( ){}
```

In C#, however, this syntax is simply a shortcut for declaring a `Finalize( )` method that chains up to its base class. Thus, writing:

```
~MyClass( )
{
   // do work here
}
```

is identical to writing:

```
MyClass.Finalize( )
{
   // do work here
   base.Finalize( );
}
```

Because of the potential for confusion, it is recommended that you eschew the destructor and write an explicit finalizer if needed.

4.4.2 Finalize Versus Dispose

It is not legal to call a finalizer explicitly. Your `Finalize( )` method will be called by the garbage collector. If you do handle precious unmanaged resources (such as file handles) that you want to close and dispose of as quickly as possible, you ought to implement the `IDisposable` interface. (You will learn more about interfaces in Chapter 8.) The `IDisposable` interface requires its implementers to define one method, named `Dispose( )`, to perform whatever cleanup you consider to be crucial. The availability of `Dispose( )` is a way for your clients to say "don't wait for `Finalize( )` to be called, do it right now."

If you provide a `Dispose( )` method, you should stop the garbage collector from calling `Finalize( )` on your object. To stop the garbage collector, you call the static method `GC.SuppressFinalize()`, passing in the `this` pointer for your object. Your `Finalize( )` method can then call your `Dispose( )` method. Thus, you might write:

```
public void Dispose(  )
{
  // perform clean up

  // tell the GC not to finailze
  GC.SuppressFinalize(this);
}

public override void Finalize(  )
{
  Dispose(  );
  base.Finalize(  );
}
```

4.4.3 Implementing the Close Method

For some objects, you'd rather have your clients call the `Close( )` method. (For example, `Close` makes more sense than `Dispose( )` for file objects.) You can implement this by creating a private `Dispose( )` method and a public `Close( )` method and having your `Close( )` method invoke `Dispose( )`.

4.4.4 The using Statement

Because you cannot be certain that your user will call `Dispose( )` reliably, and because finalization is nondeterministic (i.e., you can't control when the GC will run), C# provides a `using` statement which ensures that `Dispose( )` will be called at the earliest possible time. The idiom is to declare which objects you are using and then to create a scope for these objects with curly braces. When the close brace is reached, the `Dispose( )` method will be called on the object automatically, as illustrated in Example 4-6.

Example 4-6. The using construct

```
using System.Drawing;
class Tester
{
   public static void Main(  )
   {
      using (Font theFont = new Font("Arial", 10.0f))
      {
         // use theFont

      }   // compiler will call Dispose on theFont

      Font anotherFont = new Font("Courier",12.0f);

      using (anotherFont)
      {
         // use anotherFont

      }  // compiler calls Dispose on anotherFont

   }

}
```

In the first part of this example, the `Font` object is created within the `using` statement. When the `using` statement ends, `Dispose( )` is called on the `Font` object.

In the second part of the example, a `Font` object is created outside of the `using` statement. When we decide to use that font, we put it inside the `using` statement and when that statement ends, once again `Dispose( )` is called.

The `using` statement also protects you against unanticipated exceptions. No matter how control leaves the `using` statement, `Dispose( )` is called. It is as if there were an implicit *try-catch-finally* block. (See Section 11.2 in Chapter 11 for details.)

4.5 Passing Parameters

By default, value types are passed into methods by value (see Section 4.1.2 earlier in this chapter). This means that when a value object is passed to a method, a temporary copy of the object is created within that method. Once the method completes, the copy is discarded. Although passing by value is the normal case, there are times when you will want to pass value objects by reference. C# provides the `ref` parameter modifier for passing value objects into a method by reference and the `out` modifier for those cases in which you want to pass in a `ref` variable without first initializing it. C# also supports the `params` modifier which allows a method to accept a variable number of parameters. The `params` keyword is discussed in Chapter 9.

4.5.1 Passing by Reference

Methods can return only a single value (though that value can be a collection of values). Let's return to the `Time` class and add a `GetTime( )` method which returns the hour, minutes, and seconds.

Because we cannot return three values, perhaps we can pass in three parameters, let the method modify the parameters, and examine the result in the calling method. Example 4-7 shows a first attempt at this.

Example 4-7. Returning values in parameters

```
public class Time
{
   // public accessor methods
   public void DisplayCurrentTime(  )
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   public int GetHour(  )
   {
      return Hour;
   }

     public void GetTime(int h, int m, int s)
     {
         h = Hour;
         m = Minute;
         s = Second;
     }

   // constructor
   public Time(System.DateTime dt)
   {
         Year = dt.Year;
         Month = dt.Month;
         Date = dt.Day;
         Hour = dt.Hour;
         Minute = dt.Minute;
         Second = dt.Second;
      }

       // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second;

   }

   public class Tester
   {
      static void Main(  )
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time(currentTime);
         t.DisplayCurrentTime(  );

            int theHour = 0;
            int theMinute = 0;
            int theSecond = 0;
            t.GetTime(theHour, theMinute, theSecond);
         System.Console.WriteLine("Current time: {0}:{1}:{2}",
                theHour, theMinute, theSecond);

      }

   }
```

```
 Output:
11/17/2000 13:41:18
Current time: 0:0:0
```

Notice that the "Current time" in the output is `0:0:0`. Clearly, this first attempt did not work. The problem is with the parameters. We pass in three integer parameters to `GetTime( )`, and we modify the parameters in `GetTime( )`, but when the values are accessed back in `Main( )`, they are unchanged. This is because integers are value types, and so are passed by value; a copy is made in `GetTime( )`. What we need is to pass these values by reference.

Two small changes are required. First, change the parameters of the `GetTime` method to indicate that the parameters are `ref` (reference) parameters:

```
public void GetTime(ref int h, ref int m, ref int s)
{
     h = Hour;
     m = Minute;
     s = Second;
}
```

Second, modify the call to `GetTime( )` to pass the arguments as references as well:

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
```

If you leave out the second step of marking the arguments with the keyword `ref`, the compiler will complain that the argument cannot be converted from an `int` to a `ref int`.

The results now show the correct time. By declaring these parameters to be `ref` parameters, you instruct the compiler to pass them by reference. Instead of a copy being made, the parameter in `GetTime( )` is a reference to the same variable (`theHour)` that is created in `Main( )`. When you change these values in `GetTime( )`, the change is reflected in `Main( )`.

Keep in mind that ref parameters are references to the actual original value—it is as if you said "here, work on this one." Conversely, value parameters are copies—it is as if you said "here, work on one *just like* this."

4.5.2 Passing Out Parameters with Definite Assignment

C# imposes *definite assignment* , which requires that all variables be assigned a value before they are used. In Example 4-7, if you don't initialize `theHour`, `theMinute`, and `theSecond` before you pass them as parameters to `GetTime( )`, the compiler will complain. Yet the initialization that is done merely sets their values to `0` before they are passed to the method:

```
int theHour = 0;
int theMinute = 0;
int theSecond = 0;
t.GetTime( ref theHour, ref theMinute, ref theSecond);
```

It seems silly to initialize these values because you immediately pass them by reference into `GetTime` where they'll be changed, but if you don't, the following compiler errors are reported:

```
Use of unassigned local variable 'theHour'
Use of unassigned local variable 'theMinute'
Use of unassigned local variable 'theSecond'
```

C# provides the `out` parameter modifier for this situation. The `out` modifier removes the requirement that a reference parameter be initiailzed. The parameters to `GetTime( )`, for example, provide no information to the method; they are simply a mechanism for getting information out of it. Thus, by marking all three as `out` parameters, you eliminate the need to initialize them outside the method. Within the called method the `out` parameters must be assigned a value before the method returns. Here are the altered parameter declarations for `GetTime( )`:

```
public void GetTime(out int h, out int m, out int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
```

and here is the new invocation of the method in `Main( )`:

```
t.GetTime( out theHour, out theMinute, out theSecond);
```

To summarize, value types are passed into methods by value. `Ref` parameters are used to pass value types into a method by reference. This allows you to retrieve their modified value in the calling method. `Out` parameters are used only to return information from a method. Example 4-8 rewrites Example 4-7 to use all three.

Example 4-8. Using in, out, and ref parameters

```
public class Time
  {
     // public accessor methods
     public void DisplayCurrentTime(  )
     {
        System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
           Month, Date, Year, Hour, Minute, Second);
     }

     public int GetHour(  )
     {
        return Hour;
     }

       public void SetTime(int hr, out int min, ref int sec)
       {
           // if the passed in time is >= 30
           // increment the minute and set second to 0
           // otherwise leave both alone
           if (sec >= 30)
           {
               Minute++;
               Second = 0;
           }
           Hour = hr; // set to value passed in

           // pass the minute and second back out
           min = Minute;
           sec = Second;
       }

     // constructor
     public Time(System.DateTime dt)
     {

        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
     }

      // private member variables
     private int Year;
     private int Month;
     private int Date;
     private int Hour;
     private int Minute;
     private int Second;

  }

  public class Tester
  {
     static void Main(  )
     {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
      Hour =      dt.Hour;
      Minute =    dt.Minute;
      Second =    dt.Second;
   }


   // private member variables
   public static int Year;
   public static int Month;
   public static int Date;
   public static int Hour;
   public static int Minute;
   public static int Second;
}

public class Tester
{
   static void Main(  )
   {
      System.Console.WriteLine ("This year: {0}",
             RightNow.Year.ToString(  ));
      RightNow.Year = 2002;
      System.Console.WriteLine ("This year: {0}",
      RightNow.Year.ToString(  ));
   }
}
```

Output:

```
This year: 2000
This year: 2002
```

This works well enough, until someone comes along and changes one of these values. As the example shows, the `RightNow.Year` value can be changed, for example, to `2002`. This is clearly not what we'd like.

We'd like to mark the static values as constant, but that is not possible because we don't initialize them until the static constructor is executed. C# provides the keyword `readonly` for exactly this purpose. If you change the class member variable declarations as follows:

```
public static readonly int Year;
public static readonly int Month;
public static readonly int Date;
public static readonly int Hour;
public static readonly int Minute;
public static readonly int Second;
```

then comment out the reassignment in `Main( )`:

```
// RightNow.Year = 2002; // error!
```

the program will compile and run as intended.

Appendix B. Conversions

Overview

In C#, conversions are divided into implicit and explicit conversions. Implicit conversions are those that will always succeed; the conversion can always be performed without data loss.[1] For numeric types, this means that the destination type can fully represent the range of the source type. For example, a `short` can be converted implicitly to an `int`, because the `short` range is a subset of the `int` range.

[1]Conversions from `int`, `uint`, or `long` to `float` and from `long` to `double` may result in a loss of precision, but will not result in a loss of magnitude.

Numeric Types

For the numeric types, there are widening implicit conversions for all the signed and unsigned numeric types. Figure 15-1 shows the conversion hierarchy. If a path of arrows can be followed from a source type to a destination type, there is an implicit conversion from the source to the destination. For example, there are implicit conversions from `sbyte` to `short`, from `byte` to `decimal`, and from `ushort` to `long`.

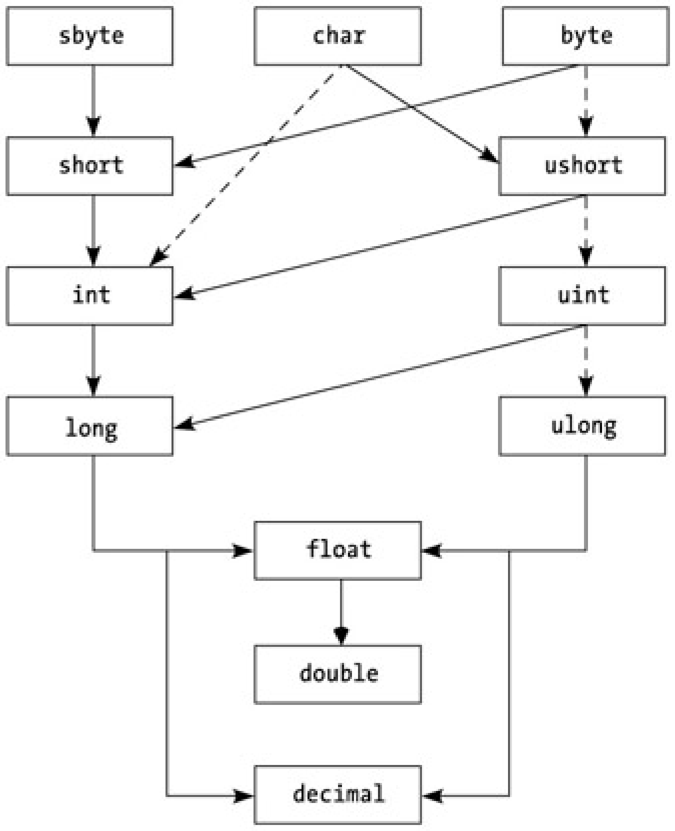


Figure 15-1. C# conversion hierarchy

Note that the path taken from a source type to a destination type in the figure does not represent how the conversion is done; it merely indicates that it can be done. In other words, the conversion from `byte` to `long` is done in a single operation, not by converting through `ushort` and `uint`.

class Test

```
{
    public static void Main()
    {
            // all implicit
        sbyte v = 55;
        short v2 = v;
        int v3 = v2;
        long v4 = v3;

            // explicit to "smaller" types
        v3 = (int) v4;
        v2 = (short) v3;
        v = (sbyte) v2;
    }
}
```

Conversions and Member Lookup

When considering overloaded members, the compiler may have to choose between several functions. Consider the following:

```
using System;
class Conv
{
    public static void Process(sbyte value)
    {
        Console.WriteLine("sbyte {0}", value);
    }
    public static void Process(short value)
    {
        Console.WriteLine("short {0}", value);
    }
    public static void Process(int value)
    {
        Console.WriteLine("int {0}", value);
    }
}
class Test
{
     public static void Main()
     {
         int value1 = 2;
         sbyte value2 = 1;
         Conv.Process(value1);
         Conv.Process(value2);
     }
}
```

The preceding code produces the following output:

```
int 2
sbyte 1
```

In the first call to `Process()`, the compiler could only match the `int` parameter to one of the functions, the one that took an `int` parameter.
In the second call, however, the compiler had three versions to choose from, taking `sbyte`, `short`, or `int`. To select one version, it first tries to match the type exactly. In this case, it can match `sbyte`, so that's the version that gets called. If the `sbyte` version wasn't there, it would select the `short` version, because a `short` can be converted implicitly to an `int`. In other words, `short` is "closer to" `sbyte` in the conversion hierarchy, and is therefore preferred.

The preceding rule handles many cases, but it doesn't handle the following one:

```
using System;
class Conv
{
   public static void Process(short value)
   {
      Console.WriteLine("short {0}", value);
   }
   public static void Process(ushort value)
   {
      Console.WriteLine("ushort {0}", value);
   }
}
class Test
{
   public static void Main()
   {
      byte value = 3;
      Conv.Process(value);
   }
}
```

Here, the earlier rule doesn't allow the compiler to choose one function over the other, because there are no implicit conversions in either direction between `ushort` and `short`.
In this case, there's another rule that kicks in, which says that if there is a single-arrow implicit conversion to a signed type, it will be preferred over all conversions to unsigned types. This is graphically represented in Figure 15-1 by the dotted arrows; the compiler will choose a single solid arrow over any number of dotted arrows.
This rule only applies for the case where there is a single-arrow conversion to the signed type. If the function that took a `short` was changed to take an `int`, there would be no "better" conversion, and an ambiguity error would be reported.

Explicit Numeric Conversions

Explicit conversions—those using the cast syntax—are the conversions that operate in the opposite direction from the implicit conversions. Converting from `short` to `long` is implicit, and therefore converting from `long` to `short` is an explicit conversion.

Viewed another way, an explicit numeric conversion may result in a value that is different than the original:

```
using System;
class Test
```

```
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2 = (byte) value1;
        Console.WriteLine("Value2: {0}", value2);
    }
}
```

The preceding code results in the following output:

```
56
```

In the conversion to `byte`, the least-significant (lowest-valued) part of the `uint` is put into the `byte` value. In many cases, the programmer either knows that the conversion will succeed, or is depending on this behavior.

Checked Conversions

In other cases, it may be useful to check whether the conversion succeeded. This is done by executing the conversion in a `checked` context:

```
using System;
class Test
{
    public static void Main()
    {
        checked
        {
            uint value1 = 312;
            byte value2 = (byte) value1;
            Console.WriteLine("Value: {0}", value2);
        }
    }
}
```

When an explicit numeric conversion is done in a `checked` context, if the source value will not fit in the destination data type, an exception will be thrown.

The `checked` statement creates a block in which conversions are checked for success. Whether a conversion is checked or not is determined at compile time, and the checked state does not apply to code in functions called from within the `checked` block.

Checking conversions for success does have a small performance penalty, and therefore may not be appropriate for released software. It can, however, be useful to check all explicit numeric conversions when developing software. The C# compiler provides a `/checked` compiler option that will generate checked conversions for all explicit numeric conversions. This option can be used while developing software, and then can be turned off to improve performance for released software.

If the programmer is depending upon the unchecked behavior, turning on `/checked` could cause problems. In this case, the `unchecked` statement can be used to indicate that none of the conversions in a block should ever be checked for conversions.

It is sometimes useful to be able to specify the checked state for a single statement; in this case, the `checked` or `unchecked` operator can be specified at the beginning of an expression:

```
using System;
class Test
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2;

        value2 = unchecked((byte) value1);   // never checked
        value2 = (byte) value1;              // checked if /checked
        value2 = checked((byte) value1);     // always checked
    }
}
```
In this example, the first conversion will never be checked, the second conversion will be checked if the `/checked` statement is present, and the third conversion will always be checked.

Conversions of Classes (Reference Types)

Conversions involving classes are similar to those involving numeric values, except that object conversions deal with casts up and down the object inheritance hierarchy instead of conversions up and down the numeric type hierarchy.

As with numeric conversions, implicit conversions are those that will always succeed, and explicit conversions are those that may fail.

To the Base Class of an Object

A reference to an object can be converted implicitly to a reference to the base class of an object. Note that this does *not* convert the object to the type of the base class; only the reference is to the base class type. The following example illustrates this:

```
using System;
public class Base
{
    public virtual void WhoAmI()
    {
        Console.WriteLine("Base");
    }
}
public class Derived: Base
{
    public override void WhoAmI()
    {
        Console.WriteLine("Derived");
    }
}
public class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        Base b = d;

        b.WhoAmI();
        Derived d2 = (Derived) b;
        Object o = d;
        Derived d3 = (Derived) o;
    }
}
```

This code produces the following output:

```
Derived
```

Initially, a new instance of `Derived` is created, and the variable `d` contains a reference to that object. The reference `d` is then converted to a reference to the base type `Base`. The object referenced by both variables, however, is still a `Derived` ; this is shown because when the virtual function `WhoAmI()` is called, the version from `Derived` is called. It is also possible to convert the `Base` reference `b` back to a reference of type `Derived`, or to convert the `Derived` reference to an `object` reference and back.
Converting to the base type is an implicit conversion because, as discussed in Chapter 1, "Object-Oriented Basics," a derived class *is* always an example of the base class. In other words, `Derived` is-a `Base`.
Explicit conversions are possible between classes when there is a "could-be" relationship. Because `Derived` is derived from `Base`, any reference to `Base` could really be a `Base` reference to a `Derived` object, and therefore the conversion can be attempted. At runtime, the actual type of the object referenced by the `Base` reference (in the previous example) will be checked to see if it is really a reference to `Derived`. If it isn't, an exception will be thrown on the conversion.
Because `object` is the ultimate base type, any reference to a class can be implicitly converted to a reference to `object`, and a reference to `object` may be explicitly converted to a reference to any class type.
Figure 15-2 shows the previous example pictorially.

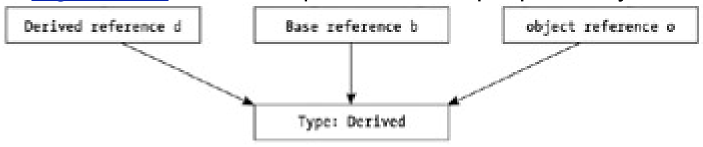


Figure 15-2. Different references to the same instance

To an Interface the Object Implements

Interface implementation is somewhat like class inheritance. If a class implements an interface, an implicit conversion can be used to convert from a reference to an instance of the class to the interface. This conversion is implicit because it is known at compile time that it works.

Once again, the conversion to an interface does not change the underlying type of an object. A reference to an interface can therefore be converted explicitly back to a reference to an object that implements the interface, since the interface reference "could-be" referencing an instance of the specified object.

In practice, converting back from the interface to an object is an operation that is rarely, if ever, used.

To an Interface the Object Might Implement

The implicit conversion from an object reference to an interface reference discussed in the previous section isn't the common case. An interface is especially useful in situations where it isn't known whether an object implements an interface.

The following example implements a debug trace routine that uses an interface if it's available:

```
using System;
interface IdebugDump
{
   string DumpObject();
}
class Simple
```

```
{
    public Simple(int value)
    {
        this.value = value;
    }
    public override string ToString()
    {
        return(value.ToString());
    }
    int value;
}
class Complicated: IdebugDump
{
    public Complicated(string name)
    {
        this.name = name;
    }
    public override string ToString()
    {
        return(name);
    }
    string IdebugDump.DumpObject()
    {
        return(String.Format(
            "{0}\nLatency: {0}\nRequests: {1}\nFailures; {0}\n",
            new object[] {name,    latency, requestCount, failedCount} ));
    }
    string name;
    int latency = 0;
    int requestCount = 0;
    int failedCount = 0;
}
    class Test
    {
        public static void DoConsoleDump(params object[] arr)
        {
            foreach (object o in arr)
            {
                IDebugDump dumper = o as IDebugDump;
                if (dumper != null)
                    Console.WriteLine("{0}", dumper.DumpObject());
                else
                    Console.WriteLine("{0}", o);
            }
```

Appendix C: Math Class

Working with numbers is the most fundamental programming task. The Microsoft .NET Framework and C# add a few features to numbers that may be new to veteran C programmers. In this appendix, I'll discuss those features as well as the all-important *Math* class, which contains methods that are equivalents of functions declared in the C Math.h header file.

Numeric Types

The C# language supports 11 numeric types that fall into three categories: integral, floating point, and decimal:

C# Numeric Types

	Integers			
Bits	**Signed**	**Unsigned**	**Floating Point**	**Decimal**
8	*sbyte*	*byte*		
16	*short*	*ushort*		
32	*int*	*uint*	*float*	
64	*long*	*ulong*	*double*	
128				*decimal*

In a C# program, an integer literal (that is, a number written without a decimal point) is assumed to be an *int* unless its value is larger than a maximum *int*, in which case the value of the number is used to determine its type. The number is assumed to be a *uint*, *long*, or *ulong* (in that order) depending on its value. A literal with a decimal point (or that includes an exponent indicated with an *E* or *e* followed by a number) is assumed to be a *double*. You can use the following suffixes on numeric literals to clarify your intentions.

Suffixes for Numeric Literals

Type	**Suffix**
uint	*u* or *U*
long	*l* or *L*
ulong	*ul*, *uL*, *Ul*, *UL*, *lu*, *lU*, *Lu*, or *LU*
float	*f* or *F*
double	*d* or *D*
decimal	*m* or *M*

The C# type names are aliases for structures defined in the *System* class of the .NET Framework. These structures are all derived from *ValueType*, which itself derives from *Object*:

.NET Numeric Types

	Integers			
Bits	**Signed**	**Unsigned**	**Floating Point**	**Decimal**
8	*SByte*	*Byte*		
16	*Int16*	*UInt16*		
32	*Int32*	*UInt32*	*Single*	
64	*Int64*	*UInt64*	*Double*	
128				*Decimal*

The *SByte*, *UInt16*, *UInt32*, and *UInt64* types are not compliant with the Common Language Specification (CLS). What that means is that a programming language can be compliant with the CLS without supporting these types. If you write code that you want to be usable by all CLS-compliant languages (such as in DLLs), do not use signed bytes or unsigned 16-bit, 32-bit, or 64-bit integers.

Checking Integer Overflow

Consider the following code:

```
short s = 32767;

s += 1;
```

Here's another one:

```
ushort us = 0;

us -= 1;
```

These are examples of integer overflow and underflow, and both these snippets of code are perfectly legal in C as well as C# (by default anyway).

In the first case, a signed integer is being incremented past its maximum value. Due to the manner in which integers are stored in memory, the result will be −32768. In the second case, an unsigned integer is being decremented below zero, and the result is 65535.

Sometimes programmers take advantage of integer overflow and underflow, and sometimes programmers fall victim to overflow and underflow bugs. To separate clever techniques from nasty bugs, C# allows you to optionally check for integer overflow and underflow.

To subject an entire program to runtime checking of overflow and underflow, use the following compiler switch:

```
/checked+
```

The following compiler switch results in the default option:

```
/checked-
```

In Visual C# .NET, you can set this compiler switch by first invoking the Property Pages dialog box for the project. On the left side of the dialog box, select Build from Configuration Properties. On the right side of the dialog box, set the option Check For Arithmetic Overflow/Underflow to True.

When you enable runtime checking of overflow and underflow, the increment and decrement operations just shown will raise an exception of type *OverflowException*.

Within your C# program, you can override the compiler setting by using the keywords *checked* and *unchecked*. You follow the keyword with an expression enclosed in parentheses, or a statement or group of statements in curly brackets. For example, the code

```
short s = 32767;

checked

{

     s += 1;

}
```

will raise an exception regardless of the compiler switch. You'll probably want to enclose *checked* blocks within *try* blocks.

So far, I've been speaking solely of *runtime* checking of integer overflow and underflow. By default, the compiler will flag compile-time overflow and underflow as an error regardless of the compiler switch you use. For example, the statement

```
short s = 32767 + 1;
```

is always a compile-time error because the addition is evaluated during compilation. However, it is possible to use the *unchecked* keyword to override compile-time overflow and underflow checking. For example, suppose you define two *const* integers like so:

```
const int i1 = 65536;

const int i2 = 65536;
```

The expression

```
int i3 = i1 * i2;
```

will normally cause a compile-time error. Because *i1* and *i2* are both *const* values, the compiler attempts to evaluate the expression and encounters an overflow. A compiler switch won't override that behavior, but the *unchecked* keyword will:

```
int i3 = unchecked (i1 * i2);
```

That statement will compile fine and execute without raising an exception.

You should probably write your program to compile and run correctly under either compiler option. Whenever there's a danger of overflow or underflow that you want to catch, enclose the statement in a *checked* block within a *try* block. Whenever you don't care about overflow or underflow, or you want to exploit overflow or underflow in some way, enclose the statement in an *unchecked* block.

Regardless of any compiler switches or the presence of the *checked* and *unchecked* keywords, integer division by zero always raises a *DivideByZeroException*.

The Decimal Type

The C# numeric type that is entirely new to C programmers is the *decimal* type, which uses 16 bytes (128 bits) to store each value. The 128 bits break down into a 96-bit integral part, a 1-bit sign, and a scaling factor that can range from 0 through 28. Mathematically, the scaling factor is a negative power of 10 and indicates the number of decimal places in the number.

Don't confuse the *decimal* type with a *binary-coded decimal* (BCD) type. In a BCD type, each decimal digit is stored using 4 bits. The *decimal* type stores the number as a binary integer.

For example, if you define a *decimal* equal to 12.34, the number is stored as the integer 0x4D2 (or 1234) with a scaling factor of 2. A BCD encoding would store the number as 0x1234.

As long as a decimal number has 28 significant digits (or fewer) and 28 decimal places (or fewer), the *decimal* data type stores the number exactly. This is not true with floating point! If you define a *float* equal to 12.34, it's essentially stored as the value 0xC570A4 (or 12,939,428) divided by 0x100000 (or 1,048,576), which is only *approximately* 12.34. Even if you define a *double* equal to 12.34, it's stored as the value 0x18AE147AE147AE (or 6,946,802,425,218,990) divided by 0x20000000000000 (or 562,949,953,421,312), which again only approximately equals 12.34.

And that's why you should use *decimal* when you're performing calculations where you don't want pennies to mysteriously crop up and disappear. The floating-point data type is great for scientific and engineering applications but often undesirable for financial ones.

If you want to explore the internals of the *decimal*, you can make use of the following constructor:

Decimal **Constructors (selection)**

```
Decimal(int iLow, int iMiddle, int iHigh, bool bNegative, byte byScale)
```

Although defined as integers, the first three arguments of the constructor are treated as unsigned integers to form a composite 96-bit unsigned integer. The *byScale* argument (which can range from 0 through 28) is the number of decimal places. For example, the expression

```
new Decimal(123456789, 0, 0, false, 5)
```

creates the *decimal* number

```
1234.56789
```

The largest positive *decimal* number is

```
new Decimal(-1, -1, -1, false, 0)
```

or

```
79,228,162,514,264,337,593,543,950,335
```

which you can also obtain from the *MaxValue* field of the *Decimal* structure:

```
Decimal.MaxValue
```

The smallest *decimal* number closest to 0 is

```
new Decimal(1, 0, 0, false, 28)
```

which equals

```
0.0000000000000000000000000001
```

or

1×10^{-28}

If you divide this number by 2 in a C# program, the result is 0.

It's also possible to obtain the bits used to store a *decimal* value:

Decimal Static Methods (selection)

```
int[] GetBits(decimal mValue)
```

This method returns an array of four integers. The first, second, and third elements of the array are the low, medium, and high components of the 96-bit unsigned integer. The fourth element contains the sign and the scaling factor: bits 0 through 15 are 0; bits 16 through 23 contain a scaling value between 0 and 28; bits 24 through 30 are 0; and bit 31 is 0 for positive and 1 for negative.

If you have a *decimal* number named *mValue*, you can execute the statement

```
int[] ai = Decimal.GetBits(mValue);
```

If *ai[3]* is negative, the *decimal* number is negative. The scaling factor is

```
(ai[3] >> 16) & 0xFF
```

I already indicated how floating-point representation is often only approximate. When you start performing arithmetic operations on floating-point numbers, the approximations can get worse. Almost everyone who has used floating point is well aware that a number that should be 4.55 (for example) is often stored as 4.549999 or 4.550001.

The decimal representation is much better behaved. For example, suppose *m1* is defined like so:

```
decimal m1 = 12.34;
```

Internally, *m1* has an integer part of 1234 and a scaling factor of 2. Also, suppose *m2* is defined like this:

```
decimal m2 = 56.789;
```

The integer part is 56789, and the scaling factor is 3. Now add these two numbers:

```
decimal m3 = m1 + m2;
```

Internally, the integer part of *m1* is multiplied by 10 (to get 12340), and the scaling factor is set to 3. Now the integer parts can be added directly: 12340 plus 56789 equals 69129 with a scaling factor of 3. The actual number is 69.129. Everything is exact.

Now multiply the two numbers:

```
decimal m4 = m1 * m2;
```

Internally, the two integral parts are multiplied (1234 times 56789 equals 70,077,626), and the scaling factors are added (2 plus 3 equals 5). The actual numeric result is 700.77626. Again, the calculation is exact.

When dividing…well, division is messy no matter how you do it. But for the most part, when using *decimal*, you have much better control over the precision and accuracy of your results.

Floating-Point Infinity and NaNs

The two floating-point data types—*float* and *double*—are defined in accordance with the ANSI/IEEE Std 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*.

A *float* value consists of a 24-bit signed mantissa and an 8-bit signed exponent. The precision is approximately seven decimal digits. Values range from

-3.402823×10^{38}

to

3.402823×10^{38}

The smallest possible *float* value greater than 0 is

1.401298×10^{-45}

You can obtain these three values as fields in the *Single* structure:

***Single* Structure Constant Fields (selection)**

Type	Field
float	*MinValue*
float	*MaxValue*
float	*Epsilon*

A *double* value consists of a 53-bit signed mantissa and an 11-bit signed exponent. The precision is approximately 15 to 16 decimal digits. Values range from

$-1.79769313486232 \times 10^{308}$

to

$1.79769313486232 \times 10^{308}$

The smallest possible *double* value greater than 0 is

$4.94065645841247 \times 10^{-324}$

The *MinValue*, *MaxValue*, and *Epsilon* fields are also defined in the *Double* structure.

Here's some code that divides a floating-point number by 0:

```
float f1 = 1;
float f2 = 0;
float f3 = f1 / f2;
```

If these were integers, a *DivideByZeroException* would be raised. But these are IEEE floating-point numbers. An exception is *not* raised. Indeed, floating-point operations *never* raise exceptions. Instead, in this case, *f3* takes on a special value. If you use *Console.WriteLine* to display *f3*, it will display the word

```
Infinity
```

If you change the initialization of *f1* to –1, *Console.WriteLine* will display

```
-Infinity
```

In the IEEE standard, positive infinity and negative infinity are legitimate values of floating-point numbers. You can even perform arithmetic on infinite values. For example, the expression

```
1 / f3
```

equals 0.

If you change the initialization of *f1* in the preceding code to 0, then *f3* will equal a value known as *Not a Number*, which is universally abbreviated as *NaN* and pronounced "nan." Here's how *Console.WriteLine* displays a NaN:

```
NaN
```

You can also create a NaN by adding a positive infinity to a negative infinity or by a number of other calculations.

Both the *Single* and *Double* structures have static methods to determine whether a *float* or *double* value is infinity or NaN. Here are the methods in the *Single* structure:

Single Structure Static Methods (selection)

```
bool IsInfinity(float fValue)
bool IsPositiveInfinity(float fValue)
bool IsNegativeInfinity(float fValue)
bool IsNaN(float fValue)
```

For example, the expression

```
Single.IsInfinity(fVal)
```

returns *true* if *fVal* is either positive infinity or negative infinity.

The *Single* structure also has constant fields that represent these values:

Single Structure Constant Fields (selection)

Type	Field
float	*PositiveInfinity*
float	*NegativeInfinity*
float	*NaN*

Identical fields are defined in the *Double* structure. These values correspond to specific bit patterns defined in the IEEE standard.

The *Math* Class

The *Math* class in the *System* namespace consists solely of a collection of static methods and the following two constant fields:

Math Constant Fields

Type	Field	Value
double	*PI*	3.14159265358979
double	*E*	2.71828182845905

Math.PI, of course, is the ratio of the circumference of a circle to its diameter, and *Math.E* is the limit of

$$\left(1+\frac{1}{n}\right)^{n}$$

as *n* approaches infinity.

Most of the methods in the *Math* class are defined only for *double* values. However, some methods are defined for integer and *decimal* values as well. The following two methods are defined for every numeric type:

***Math* Static Methods (selection)**

```
type Max(numeric-type n1, numeric-type n2)
type Min(numeric-type n1, numeric-type n2)
```

The two values must be the same type.

The following two methods are defined for *float*, *double*, *decimal*, and all signed integer types:

***Math* Static Methods (selection)**

```
int Sign(signed-type s)
type Abs(signed-type s)
```

The *Sign* method returns 1 if the argument is positive, −1 if the argument is negative, and 0 if the argument is 0. The *Abs* method returns the argument if it's 0 or positive, and the negative value of the argument if the argument is negative.

The *Abs* method is the only method of the *Math* class that can throw an exception, and then only for integral arguments, and only for one particular value for each integral type. If the argument is the *MinValue* of the particular integral type (for example, –32768 for *short*), then an *OverflowException* is raised because 32768 can't be represented by a *short*.

The following methods perform various types of rounding on *double* and *decimal* values:

***Math* Static Methods (selection)**

```
double Floor(double dValue)
double Ceiling(double dValue)
double Round(double dValue)
double Round(double dValue, int iDecimals)
decimal Round(decimal mValue)
decimal Round(decimal mValue, int iDecimals)
```

Floor returns the largest whole number less than or equal to the argument; *Ceiling* returns the smallest whole number greater than or equal to the argument. The call

```
Math.Floor(3.5)
```

returns 3, and

```
Math.Ceiling(3.5)
```

returns 4. The same rules apply to negative numbers. The call

```
Math.Floor(-3.5)
```

returns –4, and

```
Math.Ceiling(-3.5)
```

returns –3.

The *Floor* method returns the nearest whole number in the direction of negative infinity, and that's why it's sometimes also known as *rounding toward negative infinity*; likewise, *Ceiling* returns the nearest whole number in the direction of positive infinity and is sometimes called *rounding toward positive infinity*. It's also possible to round toward 0, which is to obtain the nearest whole number closest to 0. You round toward 0 by casting to an integer. The expression

```
(int) 3.5
```

returns 3, and

```
(int) -3.5
```

returns –3. Rounding toward 0 is sometimes called *truncation*.

The *Round* methods with a single argument return the whole number nearest to the argument. If the argument to *Round* is midway between two whole numbers, the return value is the nearest even number. For example, the call

```
Math.Round(4.5)
```

returns 4, and

```
Math.Round(5.5)
```

returns 6. You can optionally supply an integer that indicates the number of decimal places in the return value. For example,

```
Math.Round(5.285, 2)
```

returns 5.28.

Floating-Point Remainders

Much confusion surrounds functions that calculate floating-point remainders. The C# remainder or modulus operator (%) is defined for all numeric types. (In C, the modulus operator is not defined for *float* and *double*, the *fmod* function must be used instead.) Here's a C# statement using *float* numbers with the remainder operator:

```
fResult = fDividend % fDivisor;
```

The sign of *fResult* is the same as the sign of *fDividend*, and *fResult* can be calculated with the formula

```
fResult = fDividend - n * fDivisor
```

where *n* is the largest possible integer less than or equal to *fDividend* / *fDivisor*. For example, the expression

```
4.5 % 1.25
```

equals 0.75. Let's run through the calculation. The expression 4.5 / 1.25 equals 3.6, so *n* equals 3. The quantity 4.5 minus (3 times 1.25) equals 0.75.

The IEEE standard defines a remainder a little differently, where *n* is the integer *closest* to *fDividend* / *fDivisor*. You can calculate a remainder in accordance with the IEEE standard using this method:

Math Static Methods (selection)

```
double IEEERemainder(double dDividend, double dDivisor)
```

The expression

```
Math.IEEERemainder(4.5, 1.25)
```

returns –0.5. That's because 4.5 / 1.25 equals 3.6, and the closest integer to 3.6 is 4. When *n* equals 4, the quantity 4.5 minus (4 times 1.25) equals –0.5.

Powers and Logarithms

Three methods of the *Math* class involve powers:

***Math* Static Methods (selection)**

```
double Pow(double dBase, double dPower)
double Exp(double dPower)
double Sqrt(double dValue)
```

Pow calculates the value

$\mathtt{dBase}^{\mathtt{dPower}}$

The expression

```
Math.Exp(dPower)
```

is equivalent to

```
Math.Pow(Math.E, dPower)
```

and the square root function

```
Math.Sqrt(dValue)
```

is equivalent to

```
Math.Pow(dValue, 0.5)
```

The *Sqrt* method returns NaN if the argument is negative.

The *Math* class has three methods that calculate logarithms:

***Math* Static Methods (selection)**

```
double Log10(double dValue)
double Log(double dValue)
double Log(double dValue, double dBase)
```

The expression

```
Math.Log10(dValue)
```

is equivalent to

```
Math.Log(dValue, 10)
```

and

```
Math.Log(dValue)
```

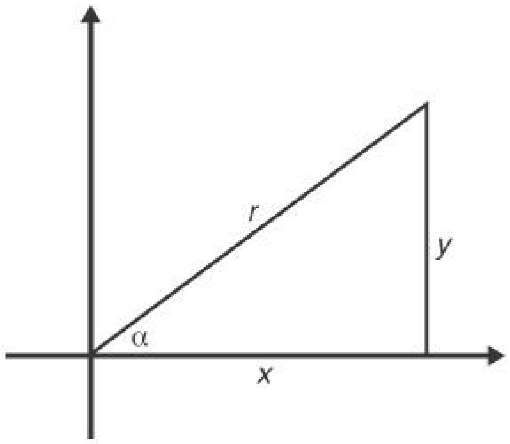
is equivalent to

```
Math.Log(dValue, Math.E)
```

The logarithm methods return *PositiveInfinity* for an argument of 0 and *NaN* for an argument less than 0.

Trigonometric Functions

Trigonometric functions describe the relationship between the sides and angles of triangles. The trig functions are defined for right triangles:



For angle α in a right triangle where *x* is the adjacent leg, *y* is the opposite leg, and *r* is the hypotenuse, the three basic trigonometric functions are

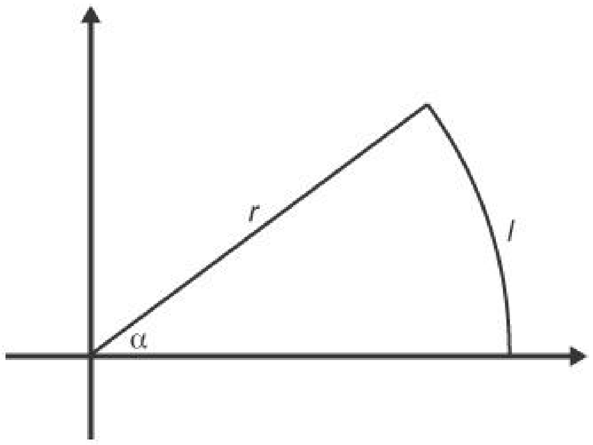
$\sin(\alpha) = y / r$
$\cos(\alpha) = x / r$
$\tan(\alpha) = y / x$

Trigonometric functions can also be used to define circles and ellipses. For constant *r* and α ranging from 0 degrees to 360 degrees, the set of coordinates (*x*, *y*) where

$x = r \cdot \sin(\alpha)$
$y = r \cdot \cos(\alpha)$

define a circle centered at the origin with radius *r*. Chapter 5 shows how to use trigonometric functions to draw circles and ellipses. Trig functions also show up in various graphics exercises in Chapters 13, 15, 17, and 19.

The trigonometric functions in the *Math* class require angles specified in radians rather than degrees. There are 2π radians in 360 degrees. The rationale for using radians can be illustrated by considering the following arc *l* subtended by angle α:



What is the length of arc *l*? Because the circumference of the entire circle equals 2πr, the length of arc *l* equals (α/360)2πr, where α is measured in degrees. However, if α is measured in radians, then the length of arc *l* simply equals αr. For a unit circle (radius equal to 1), the length of arc *l* equals the angle α in radians. And that's how the radian is defined: in a unit circle, an arc of length *l* is subtended by an angle in radians equal to *l*.

For example, an angle of 90 degrees in a unit circle subtends an arc with length π/2. Thus, 90 degrees is equivalent to π/2 radians. An angle of 180 degrees is equivalent to π radians. There are 2π radians in 360 degrees.

Here are the three basic trigonometric functions defined in the *Math* class:

***Math* Static Methods (selection)**

```
double Sin(double dAngle)
double Cos(double dAngle)
double Tan(double dAngle)
```

If you have an angle in degrees, multiply by π and divide by 180 to convert to radians:

```
Math.Sin(Math.PI * dAngleInDegrees / 180)
```

The *Sin* and *Cos* methods return values ranging from –1 to 1. In theory, the *Tan* method should return infinity at π/2 (90 degrees) and 3π/2 (270 degrees), but it returns very large values instead.

The following methods are inverses of the trigonometric functions. They return angles in radians:

***Math* Static Methods (selection)**

Method	Argument	Return Value
`double Asin(double dValue)`	–1 through 1	–π/2 through π/2
`double Acos(double dValue)`	–1 through 1	π through 0
`double Atan(double dValue)`	–∞ through ∞	–π/2 through π/2
`double Atan2(double y, double x)`	–∞ through ∞	–π through π

To convert the return value to degrees, multiply by 180 and divide by π.

The *Asin* and *Acos* methods return NaN if the argument is not in the proper range. The *Atan2* method uses the signs of the two arguments to determine the quadrant of the angle:

***Atan2* Return Values**

y Argument	*x* Argument	Return Value
Positive	Positive	0 through π/2
Positive	Negative	π/2 through π
Negative	Negative	π through 3π/2
Negative	Positive	3π/2 through 2π

Less commonly used are the hyperbolic trigonometric functions. While the common trigonometric functions define circles and ellipses, the hyperbolic trig functions define hyperbolas:

***Math* Static Methods (selection)**

```
double Sinh(double dAngle)
double Cosh(double dAngle)
double Tanh(double dAngle)
```

The angle is expressed in hyperbolic radians.

Appendix D: String Theory

Overview

Just about every programming language ever invented implements text strings a little differently. Unlike floating-point numbers, strings are not blessed (or cursed) with an industry standard. The C programming language doesn't even have a separate data type for strings. A string is simply an array of characters terminated with a zero byte. A program references the string by a pointer to the first character in the array. C programmers appreciate the ease with which strings can be mani pulated in memory. C programmers are also quite familiar with the ease in which seemingly innocent string manipulations can become nasty bugs.

In C#, the text string is its own data type named *string*, which is an alias for the class *System.String*. The *string* data type is related to the *char* data type, of course: a *string* object can be constructed from an array of characters and also converted into an array of characters. But a *string* and a *char* array are two distinct data types.

C# strings are not zero-terminated. A string has a specific length, and once a string is created, its length can't be changed. Nor can any of the individual characters that make up a string be changed. A C# string is thus said to be *immutable*. Whenever you need to change a string in some way, you must create another string. Many methods of the *String* class create new strings based on existing strings. Many methods and properties throughout the .NET Framework create and return strings.

Here's a common pitfall: you might expect that there's a method of *String* named *ToUpper* that converts all the characters in a string to uppercase, and that's precisely the case. But for a *string* instance named *str*, you can't just call the method like so:

```
str.ToUpper();   // Won't do anything!
```

Syntactically, this statement is valid, but it has no effect on the *str* variable. Strings are immutable, and hence the characters of *str* can't be altered. The *ToUpper* method creates a new string. You need to assign the return value of *ToUpper* to another string variable:

```
string strUpper = str.ToUpper();
```

Or you could assign it to the same string variable:

```
str = str.ToUpper();
```

In the second case, the original string (the one containing lowercase letters) still exists, but since it's probably no longer referenced anywhere in the program, it becomes eligible for garbage collection.

Here's another example. Suppose you define a string like so:

```
string str = "abcdifg";
```

You can access a particular character of the string by indexing the string variable:

```
char ch = str[4];
```

In this case, *ch* is the character 'i'. But you can't set a particular character of a string:

```
str[4] = 'e';  // Won't work!
```

The indexer property of the *String* class is read-only.

So, how *do* you replace characters in a C# string? There are a couple ways. The method call

```
str = str.Replace('i', 'e');
```

will replace *all* occurrences of 'i' with 'e'. Alternatively, you can first call *Remove* to create a new string with one or more characters removed at a specified index with a specified length. For example, the call

```
str = str.Remove(4, 1);
```

removes one character at the fourth position (the 'i'). You can then call *Insert* to insert a new string,

```
str = str.Insert(4, "e");
```

Or you can do both jobs in one statement:

```
str = str.Remove(4, 1).Insert(4, "e");
```

Despite the use of a single string variable named *str*, the two method calls in this last statement create two additional strings, and the quoted 'e' is yet another string.

Another approach is also possible. You can convert the string into a character array, set the appropriate element of the array, and then construct a new string based on the character array:

```
char[] ach = str.ToCharArray();

ach[4] = 'e';

str = new String(ach);
```

Or you can patch together a new string from substrings:

```
str = str.Substring(0, 4) + "e" + str.Substring(5);
```

I'll discuss all these *String* class methods more formally in the course of this appendix.

The *char* Type

Each element of a string is a *char*, which is an alias for the .NET structure *System.Char*. A program can specify a single literal character using single quotation marks:

```
char ch = 'A';
```

Although *Char* is derived from *ValueType*, a *char* variable isn't directly usable as a number. To convert a *char* variable named *ch* to an integer, for example, requires casting:

```
int i = (int) ch;
```

Character variables have numeric values from 0x0000 through 0xFFFF and refer to characters in the Unicode character set. The book *The Unicode Standard Version 3.0* (Addison-Wesley, 2000) is the essential reference to Unicode.

As in C, the backslash (\) is a special *escape* character. The character following the backslash has a special interpretation, as shown in the following table:

C# Control Characters

Character	Meaning	Value
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\t	Tab	0x0009
\n	New line	0x000A
\v	Vertical tab	0x000B
\f	Form feed	0x000C
\r	Carriage return	0x000D
\"	Double quote	0x0022
\'	Single quote	0x0027
\\	Backslash	0x005C

In addition, you can specify a single Unicode character using the preface *x* or *u* followed by a four-digit hexadecimal number. The characters '\x03A9' and '\u03A9' both refer to the Greek capital omega (Ω).

In C, as you know, you can use functions defined in the ctype.h header file to determine whether a particular character is a letter, number, control character, or whatever. In C#, you use static methods defined in the *Char* structure. The argument is either a character or a string with an index value.

Char Static Methods (selection)

```
bool IsControl(char ch)
bool IsControl(string str, int iIndex)
bool IsSeparator(char ch)
bool IsSeparator(string str, int iIndex)
bool IsWhiteSpace(char ch)
bool IsWhiteSpace(string str, int iIndex)
bool IsPunctuation(char ch)
bool IsPunctuation(string str, int iIndex)
bool IsSymbol(char ch)
bool IsSymbol(string str, int iIndex)
bool IsDigit(char ch)
bool IsDigit(string str, int iIndex)
bool IsNumber(char ch)
bool IsNumber(string str, int iIndex)
bool IsLetter(char ch)
bool IsLetter(string str, int iIndex)
bool IsUpper(char ch)
bool IsUpper(string str, int iIndex)
bool IsLower(char ch)
bool IsLower(string str, int iIndex)
bool IsLetterOrDigit(char ch)
bool IsLetterOrDigit(string str, int iIndex)
bool IsSurrogate(char ch)
bool IsSurrogate(string str, int iIndex)
```

The call

```
Char.IsControl(str[iIndex])
```

is equivalent to

```
Char.IsControl(str, iIndex)
```

You might be able to avoid using these methods for ASCII characters (character values 0x0000 through 0x007F), but these methods also apply to all Unicode characters. The *IsSurrogate* method refers to the area of Unicode with values 0xD800 through 0xDFFF that is reserved for expansion.

The *Char* structure also defines a couple other handy methods. One returns a member of the *UnicodeCategory* enumeration (defined in *System.Globalization*), and the other returns the numeric value of the character converted to a *double*:

Char Static Methods (selection)

```
UnicodeCategory GetUnicodeCategory(char ch)
UnicodeCategory GetUnicodeCategory(string str, int iIndex)
double GetNumericValue(char ch)
double GetNumericValue(string str, int iIndex)
```

String Constructors and Properties

In many cases, you'll define a string variable using a literal:

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

or a literal inserted right in a function call:

```
Console.WriteLine("Hello, world!");
```

or as the return value from one of the many methods that return string variables. One ubiquitous string-returning method is named *ToString* and converts an object to a string. For example, the expression

```
55.ToString();
```

returns the string "55".

If you preface a string literal with the @ sign, the backslash is not interpreted as an escape character. This technique is handy for specifying directories:

```
string str = @"c:\temp\my file";
```

To include a double quotation mark in such a string, use two double quotation marks in succession.

One of the less common methods of creating a string is by using one of the eight *String* constructors. Five of the *String* constructors involve pointers and are not compliant with the Common Language Specification (CLS). The remaining three *String* constructors create a *String* object by repeating a single character or converting from an array of characters:

String Constructors (selection)

```
String(char ch, int iCount)
String(char[] ach)
String(char[] ach, int iStartIndex, int iCount)
```

In the third constructor, *iStartIndex* is an index into the character array and *iCount* indicates a number of characters beginning at that index. The length of the resultant string will equal *iCount*.

The *String* class has just two properties, both of which are read-only:

String Properties

Type	Property	Accessibility
int	*Length*	get
char	*[]*	get

The first indicates the number of characters in the string; the second is an indexer that lets you access the individual characters of the string.

You can define a string variable without initializing it:

```
string str1;
```

Any attempt to use that string will cause the compiler to report that the string variable is unassigned. Because *String* is a reference type, you can assign a string variable the value *null*:

```
string str2 = null;
```

What the *null* value means is that no memory has been allocated for the string. Having a *null* value is different from having an empty string:

```
string str3 = "";
```

An empty string has memory allocated for the instance of the string, but the *str3.Length* property equals 0. Attempting to determine the length of a *null* string—making reference to *str2.Length*, for example—causes an exception to be thrown.

You can also initialize a string variable to an empty string using the only public field of the *String* class:

***String* Static Field**

Type	**Field**	**Accessibility**
string	*Empty*	read-only

For example,

```
string str = string.Empty;
```

You can define an array of strings like so:

```
string[] astr = new string[5];
```

An array of five strings is created, each of which is *null*. You can also create an array of initialized strings:

```
string[] astr = { "abc", "defghi", "jkl" };
```

This statement creates an array with three elements; that is, *astr.Length* returns 3. Each string has a specific length; for example, *astr[1].Length* returns 6.

The *String* class implements the *IComparable*, *ICloneable*, *IConvertible*, and *IEnumerable* interfaces, which implies that the *String* class contains certain methods defined in these interfaces. Because *String* implements the *IEnumerable* interface, you can use *String* with the *foreach* statement to enumerate the characters in a string. The statement

```
foreach (char ch in str)
{
     ⋮
}
```

is equivalent to (and quite a bit shorter than)

```
for (int i = 0; i < str.Length; i++)
{
     char ch = str[i];
     ⋮
}
```

In the *foreach* block, *ch* is read-only. In the *for* block, *ch* is not read-only but (as usual) the string characters can't be altered.

After *IEnumerable*, perhaps the next most important interface that *String* implements is *IComparable*, which means that the *String* class implements a method named *CompareTo* that lets you use arrays of strings with the *BinarySearch* and *Sort* methods defined in the *Array* class. I'll go over these methods later in this appendix.

Copying Strings

There are several ways to copy a string. Perhaps the simplest is using the equals sign:

```
string strCopy = str;
```

Like every class in the .NET Framework, the *String* class inherits the *ToString* method from *Object*. Because the *String* class implements *ICloneable*, it also implements the *Clone* method. These methods provide additional (if somewhat redundant) methods to copy strings:

***String* Methods (selection)**

```
string ToString()
object Clone()
```

If you use *Clone*, you must cast the result to a *string*:

```
string strCopy = (string) str.Clone();
```

The *String* class also implements a static method that copies a string:

String Copy Static Method

```
string Copy(string str)
```

Because *string* is an alias for *System.String*, you can preface the method name with the lowercase *string*:

```
string strCopy = string.Copy(str);
```

or with the fully qualified class name:

```
string strCopy = System.String.Copy(str);
```

If you have a *using System* statement in the program, you can prefix the method name with the uppercase *String* class name:

```
string strCopy = String.Copy(str);
```

Two of the *String* constructors convert a character array to a string. You can also convert a string back to a character array:

String Methods (selection)

```
char[] ToCharArray()
char[] ToCharArray(int iStartIndex, int iCount)
void CopyTo(int iStartIndexSrc, char[] achDst, int iStartIndexDst,
            int iCount)
```

The *ToCharArray* methods create the character array. The *iStartIndex* argument refers to a starting index in the string. To use the *CopyTo* method, the *achDst* array must already exist. The first argument is a starting index for the string; the third argument is a starting index in the character array. The *CopyTo* method is the equivalent of

```
for (int i = 0; i < iCount; i++)
     achDst[iStartIndexDst + i] = str[iStartIndexSrc + i];
```

The *Substring* methods create a new string that is a section of an existing string:

String Substring Method

```
string Substring(int iStartIndex)
string Substring(int iStartIndex, int iCount)
```

The first version returns a substring that begins at the index and continues to the end of the string.

Converting Strings

Two methods, each with two versions, convert strings to lowercase or uppercase:

***String* Methods (selection)**

```
string ToUpper()
string ToUpper(CultureInfo ci)
string ToLower()
string ToLower(CultureInfo ci)
```

The *CultureInfo* class is defined in *System.Globalization* and in this case refers to a particular language as used in a particular country.

Concatenating Strings

It's often necessary to tack together two or more strings, a process known as *string concatenation*. In C, you use the library functions *strcat* and *strncat*. For convenience, the C# addition operator is overloaded to perform string concatenation:

```
string str = str1 + str2;
```

The concatenation operator is convenient for defining a string literal that's a little too long to fit on a single line:

```
string str = "Those who profess to favor freedom and yet depreciate " +
             "agitation. . .want crops without plowing up the ground, they
" +
             "want rain without thunder and lightning. They want the ocean
" +
             "without the awful roar of its many waters. \x2014 Frederick
" +
             "Douglass";
```

You can also use the += operator to append a string to the end of an existing string:

```
str += "\r\n";
```

The *String* class also defines a static *Concat* method:

***String Concat* Static Method (selection)**

```
string Concat(string str1, string str2)
string Concat(string str1, string str2, string str3)
string Concat(string str1, string str2, string str3, string str4)
string Concat(params string[] astr)
```

Notice the *params* keyword in the last version of *Concat*. What that keyword means in this case is that you can specify either an array of strings or any number of strings. For example, if you have an array of strings defined as

```
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

and you pass that array to the *Concat* method

```
string str = string.Concat(astr);
```

the result is the string "abcdefghijklmnopqr". Alternatively, you can pass the individual strings directly to the *Concat* method:

```
string str = string.Concat("abc", "def", "ghi", "jkl", "mno", "pqr");
```

Although the *String* class defines *Concat* versions with two, three, four, or a variable number of arguments, only the version with the *params* argument is necessary. That method actually encompasses the other three methods.

Another set of *Concat* methods are the same except with *object* arguments:

String Concat Static Method (selection)

```
string Concat(object obj)
string Concat(object obj1, object obj2)
string Concat(object obj1, object obj2, object obj3)
string Concat(params object[] aobj)
```

The *object* arguments are converted to strings by the objects' *ToString* methods. The call

```
string.Concat(55, "-", 33, "=", 55 - 33)
```

creates the string "55-33=22".

It's sometimes necessary to concatenate an array of strings but with some kind of separator between each array element. You can do that using the *Join* static method:

String Join Static Method

```
string Join(string strSeparator, string[] astr)
string Join(string strSeparator, string[] astr, int iStartIndex, int
iCount)
```

For example, if you have an array of strings defined as

```
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

you might want to create a composite string with end-of-line indicators between each pair. Call

```
string str = string.Join("\r\n", astr);
```

The result is the string

```
abc\r\ndef\r\nghi\r\njkl\r\nmno\r\npqr
```

The separator is not appended following the last string.

The second version of *Join* lets you select a contiguous subset of strings from the array before joining them.

Comparing Strings

String is a class (not a structure), and *string* is a reference type (not a value type). Normally that would imply that the comparison operators (== and !=) wouldn't work correctly for strings. You'd be comparing object references rather than characters. However, the == and != operators have been redefined for strings and work as you'd expect. The expressions

```
(str == "New York")
```

and

```
(str != "New Jersey")
```

return *bool* values based on a case-sensitive character-by-character comparison.

There are several methods defined in the *String* class that return *bool* values indicating the result of a case-sensitive string comparison:

***String* Methods (selection)**

```
bool Equals(string str)
bool Equals(object obj)
bool StartsWith(string str)
bool EndsWith(string str)
```

If a string is defined as

```
string str = "The end of time";
```

then

```
str.StartsWith("The")
```

returns *true* but

```
str.StartsWith("the")
```

returns *false*.

There's also a static version of the *Equals* method:

***String* Static Methods (selection)**

```
bool Equals(string str1, string str2)
```

For example, instead of

```
if (str == "New York")
```

you can use

```
if (Equals(str, "New York"))
```

Methods like this one are provided primarily for languages that don't have operators for comparison.

The remaining comparison methods implemented in *String*, which I'll discuss momentarily, return an integer value that indicates whether one string is less than, equal to, or greater than another string:

***String* Comparison Method Return Values**

Return Value	Meaning
Negative	*str1 < str2*
Zero	*str1 == str2*
Positive	*str1 > str2*

Watch out: the comparison methods are defined as returning negative, zero, or positive integers, *not* –1, 0, or 1.

Usually if you're interested in whether one string is less than or greater than another, it's because you're sorting the strings in some way. And that implies that you probably don't want to perform a comparison based on the strict numeric values of the character codes. For example, you probably want the characters *e* and *é* to be regarded as less than *F*, despite the higher values of their character codes. Such a comparison is known as a *lexical* comparison rather than a *numeric* comparison.

Here's the relationship among a few select characters when compared numerically:

D < E < F < d < e < f < È < É < Ê < Ë < è < é < ê < ë

And here's the lexical comparison:

d < D < e < E < é < É < è < È < ê < Ê < ë < Ë < f < F

Is a lexical comparison also case insensitive? Mostly it is. For example, the string "New Jersey" is considered less than "new York" despite the lowercase 'n' in the second string. But when two strings are identical except for case, lowercase letters are considered less than uppercase letters, that is, "the" is less than "The". However, "Them" is less than "then".

In other words, by default, a lexical comparison is case sensitive only when a method must decide whether or not to return 0. Otherwise, it's case insensitive.

The lexical comparison also implies a certain relationship among letters, numbers, and other characters. In general, control characters are considered to be less than single quotes and dashes, which are less than white-space characters. Next comes punctuation and other symbols, digits (0 through 9), and finally letters. A null string is less than the empty string, which is less than any other character. For example,

"New" < "New York" < "Newark"

The nonstatic method *CompareTo* performs a lexical comparison between a string instance and an argument:

***String CompareTo* Method**

```
int CompareTo(string str2)
int CompareTo(object obj2)
```

The first string is the string object you're calling *CompareTo* on, for example,

```
str1.CompareTo(str2)
```

The *CompareTo* method with the object argument is necessary to implement the *IComparable* interface. The *CompareTo* method is used by the static *Array.BinarySearch* and *Array.Sort* methods, as I'll discuss shortly.

All the other comparison methods are static. The *CompareOrdinal* methods perform a strict numeric comparison based on the character value:

***String CompareOrdinal* Static Method**

```
int CompareOrdinal(string str1, string str2)
int CompareOrdinal(string str1, int iStartIndex1, string str2,
                   int iStartIndex2, int iCount)
```

The static *Compare* methods perform a lexical comparison:

***String Compare* Static Method**

```
int Compare(string str1, string str2)
int Compare(string str1, string str2, bool bIgnoreCase)
int Compare(string str1, string str2, bool bIgnoreCase, CultureInfo ci)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase, CultureInfo ci)
```

The *bIgnoreCase* argument affects the return value only when the two strings are the same except for case. Case-insensitive comparisons are much more useful for searching rather than sorting. The method calls

```
string.Compare("ë", "Ë")
```

and

```
string.Compare("ë", "Ë", false)
```

both return negative values, but

```
string.Compare("ë", "Ë", true)
```

returns 0. The calls

```
string.Compare("e", "ë", bIgnoreCase)
```

and

```
string.Compare("e", "Ë", bIgnoreCase)
```

always return negative values, regardless of the presence or value of the *bIgnoreCase* argument.

There is no comparison method implemented in the *String* class that reports that "André" equals "Andre".

Searching the String

The C library functions *strchr* and *strstr* search a string for the first occurrence of a specific character or another string and return a pointer to that occurrence. The C# equivalents—which are all versions of the *IndexOf* method—return an index in the source string rather than a pointer.

String IndexOf **Methods**

```
int IndexOf(char ch)
int IndexOf(char ch, int iStartIndex)
int IndexOf(char ch, int iStartIndex, int iCount)
int IndexOf(string str)
int IndexOf(string str, int iStartIndex)
int IndexOf(string str, int iStartIndex, int iCount)
```

You can search for a specific character or another string. The search is case sensitive. The method returns −1 if the character or string isn't found. You can optionally include a starting index and a character count. The return value is measured from the beginning of the string, not from the starting index.

With a string defined as

```
string str = "hello world";
```

then

```
str.IndexOf('o')
```

returns 4, and

```
str.IndexOf("wo")
```

returns 6.

You can also perform the search starting at the end of the string:

***String LastIndexOf* Methods**

```
int LastIndexOf(char ch)
int LastIndexOf(char ch, int iStartIndex)
int LastIndexOf(char ch, int iStartIndex, int iCount)
int LastIndexOf(string str)
int LastIndexOf(string str, int iStartIndex)
int LastIndexOf(string str, int iStartIndex, int iCount)
```

Although the methods search from the end of the string, the returned index is still measured from the beginning of the string. For the string shown above, the call

```
str.LastIndexOf('o')
```

returns 7, and

```
str.LastIndexOf("wo")
```

returns 6.

The following methods have a first argument that is an array of characters. The methods determine the first or last index in the string of a character that matches any character in the array:

***String* Methods (selection)**

```
int IndexOfAny(char[] ach)
int IndexOfAny(char[] ach, int iStartIndex)
int IndexOfAny(char[] ach, int iStartIndex, int iCount)
int LastIndexOfAny(char[] ach)
int LastIndexOfAny(char[] ach, int iStartIndex)
int LastIndexOfAny(char[] ach, int iStartIndex, int iCount)
```

If a character array and a string are defined like so:

```
char[] achVowel = { 'a', 'e', 'i', 'o', 'u' };
string str = "hello world";
```

then

```
str.IndexOfAny(achVowel)
```

returns 1, and

```
str.LastIndexOfAny(achVowel)
```

returns 7.

Trimming and Padding

Sometimes when processing text files (such as program source code files), it's convenient to remove *white space*, which is the nonvisible characters that separate other elements in the string. The *String* class has methods to do so. For purposes of these methods, white-space characters are assumed to be the following Unicode characters:

Unicode White-Space Characters

0x0009 (tab)	0x2003 (em space)
0x000A (line feed)	0x2004 (three-per-em space)
0x000B (vertical tab)	0x2005 (four-per-em space)
0x000C (form feed)	0x2006 (six-per-em space)
0x000D (carriage return)	0x2007 (figure space)
0x0020 (space)	0x2008 (punctuation space)
0x00A0 (no-break space)	0x2009 (thin space)
0x2000 (en quad)	0x200A (hair space)
0x2001 (em quad)	0x200B (zero-width space)
0x2002 (en space)	0x3000 (ideographic space)

You can either use the predefined white-space characters or define your own array of characters.

***String* Methods (selection)**

```
string Trim()
string Trim(params char[] ach)
string TrimStart(params char[] ach)
string TrimEnd(params char[] ach)
```

To remove the predefined white-space characters from the beginning and end of a string named *str*, use

```
str.Trim()
```

or

```
str.Trim(null)
```

You can also remove the predefined white-space characters from the beginning of a string, as here:

```
str.TrimStart(null)
```

or the end, as here:

```
str.TrimEnd(null)
```

Alternatively, you can specify the characters (not necessarily white-space characters) you want removed from the beginning or end of a string. You can either define a character array and pass that to the *Trim* (or *TrimStart* or *TrimEnd*) method

```
char[] achTrim = { ' ', '-', '_' };
str.Trim(achTrim)
```

or list the characters explicitly in the method call:

```
str.Trim(' ', '-', '_');
```

Both method calls cause these three characters to be stripped from the beginning and end of the string.

You can also add spaces (or any other character) to the beginning or end of a string to achieve a specified total width:

***String* Methods (selection)**

```
string PadLeft(int iTotalLength)
string PadLeft(int iTotalLength, char ch)
string PadRight(int iTotalLength)
string PadRight(int iTotalLength, char ch)
```

String Manipulation

Here are some miscellaneous methods that let you insert one string into another, remove a range of characters, and replace a particular character or string within a string. I showed examples of all these methods at the beginning of this appendix:

***String* Methods (selection)**

```
string Insert(int iIndex, string strInsert)
string Remove(int iIndex, int iCount)
string Replace(char chOld, char chNew)
string Replace(string strOld, string strNew)
```

You may have had occasion to use the C library function *strtok*. This function is intended to break a string down into *tokens*, which are substrings delimited by certain fixed characters, usually white-space characters. In C, you call *strtok* repeatedly until the source string has no more tokens. In C#, you can do the work of *strtok* with a single call to the *Split* method:

***String Split* Method**

```
string[] Split(params char[] achSeparators)
string[] Split(params char[] achSeparators, int iReturnCount)
```

If you set the first argument to *null*, the method uses the set of white-space characters shown earlier.

Formatting Strings

As you know from Chapter 1, the first argument of the *Console.Write* or *Console.WriteLine* method can be a string that describes the formatting of the remaining arguments. If these two methods are the C# equivalent of the C *printf* function, the static *Format* method of *String* is the C# equivalent of the C *sprintf* function:

***String Format* Static Method (selection)**

```
string Format(string strFormat, object obj0)
string Format(string strFormat, object obj0, object obj1)
string Format(string strFormat, object obj0, object obj1, object obj2)
string Format(string strFormat, params object[] aobj)
```

For example, the following call to *Format*,

```
string str = String.Format("The sum of {0} and {1} is {2}", 2, 3, 2 + 3);
```

creates the string "The sum of 2 and 3 is 5".

Array Sorting and Searching

The *String* class implements the *IComparable* interface, which merely requires that it implement the following method:

IComparable Method

```
int CompareTo(object obj)
```

This method is called by two useful static methods of *Array* named *Sort* and *BinarySearch*. You can use these two methods with arrays of objects of any class that implements *IComparable*.

Here are the two basic *Sort* methods:

Array Sort Static Methods (selection)

```
void Sort(Array arr)
void Sort(Array arr, int iStartIndex, int iCount)
```

The second version allows you to use a subset of the array. Suppose you define an array of strings like so:

```
string[] astr = { "New Jersey", "New York", "new Mexico", "New Hampshire"
};
```

Notice the lowercase *n* in the third string. After calling

```
Array.Sort(astr);
```

the elements of the array are reordered to be "New Hampshire", "New Jersey", "new Mexico", and "New York". Because the *Sort* method uses the *CompareTo* method of *String*, the sorting is case insensitive. However, if the array also included "New Mexico" (with an uppercase *N*), "New Mexico" would be appear after "new Mexico" in the sorted array.

The next two versions of the *Sort* method require two corresponding arrays of equal size, optionally with a starting index and an element count:

Array Sort Static Methods (selection)

```
void Sort(Array arrKeys, Array arrItems)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, int iCount)
```

The method sorts the first array and reorders the second array accordingly. I use this version of the *Sort* method in the SysInfoReflectionStrings program in Chapter 4 to sort an array of *SystemInformation* property names stored in *astrLabels*:

```
Array.Sort(astrLabels, astrValues);
```

The corresponding array of *SystemInformation* values stored in *astrValues* is also reordered so that the array elements still correspond to each other.

If you want to perform a sort using a method other than *CompareTo*, you use one of the following *Sort* methods:

Array Sort **Static Methods (selection)**

```
void Sort(Array arr, IComparer comp)
void Sort(Array arr, int iStartIndex, int iCount, IComparer comp)
void Sort(Array arrKeys, Array arrItems, IComparer comp)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, iCount,
          IComparer comp)
```

The argument of type *IComparer* can be an instance of any class that implements the *IComparer* interface. That's not the *String* class! *String* implements the *IComparable* interface, not *IComparer*.

The *IComparer* interface is defined in the *System.Collections* namespace. A class that implements *IComparer* must define the following method:

IComparer **Method**

```
int Compare(object obj1, object obj2)
```

This method is not static, and hence, is not defined in the *String* class. (The only methods named *Compare* implemented in *String* are static methods.)

The *System.Collections* namespace contains two classes that implement *IComparer*, which are *Comparer* (to perform a case-sensitive comparison just like the default) and *CaseInsensitiveComparer* (for a case-insensitive string comparison). Both these classes have a static member named *Default* that returns an instance of the class.

For example, to perform a case-sensitive sort of the string array *astr*, call

```
Array.Sort(astr);
```

or

```
Array.Sort(astr, Comparer.Default);
```

To perform a case-insensitive sort, call

```
Array.Sort(astr, CaseInsensitiveComparer.Default);
```

The case-insensitive compare is much more useful in the *BinarySearch* method rather than the *Sort* method (or when sorting in preparation for a binary search):

Array BinarySearch **Static Method**

```
int BinarySearch(Array arr, object obj)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj)
int BinarySearch(Array arr, object obj, IComparer comp)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj,
                 IComparer comp)
```

To perform a binary search, the array must be sorted. The sorted array of four state names contains the elements

```
"New Hampshire", "New Jersey", "new Mexico", "New York"
```

The call

```
Array.BinarySearch(astr, "New York")
```

returns 3 because the string is identical to *astr[3]*. The call

```
Array.BinarySearch(astr, "New Mexico")
```

returns –4. The negative number indicates that the string isn't in the array. (Remember, by default the search is case sensitive!) The complement of the return value is 3, which means that *astr[3]* is the next highest element of the array.

The call

```
Array.BinarySearch(astr, "new Mexico"));
```

returns 2 because the argument matches *astr[2]*. The call

```
Array.BinarySearch(astr, "New Mexico", CaseInsensitiveComparer.Default));
```

performs a case-insensitive search and also returns 2.

The *StringBuilder* Class

You may wonder if there's a performance penalty associated with frequent re-creations of *String* objects. Sometimes there is. Consider the following program, which uses the += operator in 10,000 string-appending operations to construct a large string.

StringAppend.cs

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

class StringAppend
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime dt  = DateTime.Now;
          string   str = String.Empty;

          for (int i = 0; i < iIterations; i++)
               str += "abcdefghijklmnopqrstuvwxyz\r\n";

          Console.WriteLine(DateTime.Now - dt);
     }
}
```

The program calls the *Now* method of the *DateTime* class at the beginning and end to calculate an elapsed time, which is displayed in hours, minutes, seconds, and units of 100 nanoseconds. (See Chapter 10 for information about *DateTime* and related classes.) Each string-appending operation causes a new *String* object to be created, which requires another memory allocation. Each previous string is marked for garbage collection.

How fast this program runs depends on how fast your machine is. It could take about a minute or so.

A better solution in a case like this is the appropriately named *StringBuilder* class, defined in the *System.Text* namespace. Unlike the string maintained by the *String* class, the string maintained by *StringBuilder* can be altered. *StringBuilder* dynamically reallocates the memory used for the string. Whenever the size of the string is about to exceed the size of the memory buffer, the buffer is doubled in size. To convert a *StringBuilder* object to a *String* object, call the *ToString* method.

Here's a revised version of the program, which uses *StringBuilder*.

StringBuilderAppend.cs

```
//--------------------------------------------------
// StringBuilderAppend.cs © 2001 by Charles Petzold
//--------------------------------------------------
using System;
using System.Text;

class StringBuilderAppend
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime      dt = DateTime.Now;
          StringBuilder sb = new StringBuilder();

          for (int i = 0; i < iIterations; i++)
               sb.Append("abcdefghijklmnopqrstuvwxyz\r\n");

          string str = sb.ToString();

          Console.WriteLine(DateTime.Now - dt);
     }
}
```

You'll probably find that this program does its work in well under a second. It seems to run in under 1/1000 the time of the original version.

Another efficient approach is to use the *StringWriter* class defined in the *System.IO* namespace. As I mentioned in Appendix A, both *StringWriter* and *StreamWriter* (which you use for writing to text files) derive from the abstract *TextWriter* class. Like *StringBuilder*, *StringWriter* assembles a composite string. The big advantage with *StringWriter* is that you can use the whole array of *Write* and *WriteLine* methods defined in the *TextWriter* class. Here's a sample program that performs the same task as the previous two programs but using a *StringWriter* object.

StringWriterAppend.cs

```
//-------------------------------------------------
// StringWriterAppend.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.IO;

class StringWriterAppend
```

```
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime     dt = DateTime.Now;
          StringWriter sw = new StringWriter();

          for (int i = 0; i < iIterations; i++)
               sw.WriteLine("abcdefghijklmnopqrstuvwxyz");

          string str = sw.ToString();

          Console.WriteLine(DateTime.Now - dt);
     }
}
```

The speed of this program is comparable to StringBuilderAppend.

There's a lesson in all this. As operating systems, programming languages, class libraries, and frameworks provide an ever increasingly higher level of abstraction, we programmers can sometimes lose sight of all the mechanisms going on beneath the surface. What looks like a simple addition in code can actually involve many layers of low-level activity.

We may be insulated from this low-level activity, but we must train ourselves to still feel the heat. If a particular operation seems slow to you, or to require too much memory, or to involve inordinately convoluted code, try to determine why and then search for an alternative. It's likely that someone has already provided exactly what you need.

Appendix E: Files and Streams

Overview

Most file I/O support in the .NET Framework is implemented in the *System.IO* namespace. On first exploration, however—and even during subsequent forays—*System.IO* can be a forbidding place. It doesn't help to be reassured that the .NET Framework offers a rich array of file I/O classes and tools. For a C programmer whose main arsenal of file I/O tools consists of library functions such as *fopen*, *fread*, *fwrite*, and *fprintf*, the .NET file I/O support can seem excessively convoluted and complex.

This appendix is intended to provide a logical progression to guide you through *System.IO*. I want to identify the really important stuff and also let you know some of the rationale for the multitude of classes.

The .NET Framework distinguishes between files and streams. A *file* is a collection of data stored on a disk with a name and (often) a directory path. When you open a file for reading or writing, it becomes a *stream*. A stream is something on which you can perform read and write operations. But streams encompass more than just open disk files. Data coming over a network is a stream, and you can also create a stream in memory. In a console application, keyboard input and text output are also streams.

The Most Essential File I/O Class

If you learn just one class in the *System.IO* namespace, let it be *FileStream*. You use this basic class to open, read from, write to, and close files. *FileStream* inherits from the abstract class *Stream*, and many of its properties and methods are derived from *Stream*.

To open an existing file or create a new file, you create an object of type *FileStream*. These five *FileStream* constructors have a nice orderly set of overloads:

***FileStream* Constructors (selection)**

```
FileStream(string strFileName, FileMode fm)
FileStream(string strFileName, FileMode fm, FileAccess fa)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs,
           int iBufferSize)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs,
           int iBufferSize, bool bAsync)
```

There are four additional *FileStream* constructors based on the operating system file handle. Those are useful for interfacing with existing code. *FileMode*, *FileAccess*, and *FileShare* are all enumerations defined in the *System.IO* namespace.

The *FileMode* enumeration indicates whether you want to open an existing file or create a new file and what should happen when the file you want to open doesn't exist or the file you want to create already exists:

***FileMode* Enumeration**

Member	Value	Description
CreateNew	1	Fails if file exists
Create	2	Deletes file contents if file already exists
Open	3	Fails if file does not exist
OpenOrCreate	4	Creates new file if file does not exist

***FileMode* Enumeration**

Member	Value	Description
Truncate	5	Fails if file does not exist; deletes contents of file
Append	6	Fails if file is opened for reading; creates new file if file does not exist; seeks to end of file

By *fail*, I mean that the *FileStream* constructor throws an exception such as *IOException* or *FileNotFoundException*. Almost always, you should call the *FileStream* constructor in a *try* block to gracefully recover from any problems regarding the presumed existence or nonexistence of the file.

Unless you specify a *FileAccess* argument, the file is opened for both reading and writing. The *FileAccess* argument indicates whether you want to read from the file, write to it, or both:

***FileAccess* Enumeration**

Member	Value	Description
Read	1	Fails for *FileMode.CreateNew*, *FileMode.Create*, *FileMode.Truncate*, or *FileMode.Append*
Write	2	Fails if file is read-only
ReadWrite	3	Fails for *FileMode.Append* or if file is read-only

There's one case in which a *FileAccess* argument is required: when you open a file with *FileMode.Append*, the constructor fails if the file is opened for reading. Because files are opened for reading and writing by default, the following constructor always fails:

```
new FileStream(strFileName, FileMode.Append)
```

If you want to use *FileMode.Append*, you also need to include an argument of *FileAccess.Write*:

```
new FileStream(strFileName, FileMode.Append, FileAccess.Write)
```

Unless you specify a *FileShare* argument, the file is open for exclusive use by your process. No other process (or the same process) can open the same file. Moreover, if any other process already has the file open and you don't specify a *FileShare* argument, the *FileStream* constructor will fail. The *FileShare* argument lets you be more specific about file sharing:

***FileShare* Enumeration (selection)**

Member	Value	Description
None	0	Allow other processes no access to the file; default
Read	1	Allow other processes to read the file
Write	2	Allow other processes to write to the file
ReadWrite	3	Allow other processes full access to the file

When you only need to read from a file, it's common to allow other processes to read from it also; in other words, *FileAccess.Read* should usually be accompanied by *FileShare.Read*. This courtesy goes both ways: if another process has a file open with *FileAccess.Read* and *FileShare.Read*, your process won't be able to open it unless you specify both flags as well.

FileStream Properties and Methods

Once you've opened a file by creating an object of type *FileStream*, you have access to the following five properties implemented in *Stream* that the *FileStream* class overrides:

***Stream* Properties**

Type	Property	Accessibility
bool	*CanRead*	get

***Stream* Properties**

Type	Property	Accessibility
bool	*CanWrite*	get
bool	*CanSeek*	get
long	*Length*	get
long	*Position*	get/set

The first two properties depend on the *FileAccess* value you used to create the *FileStream* object. The *CanSeek* property is always *true* for open files. The property can return *false* for other types of streams (such as network streams).

The *Length* and *Position* properties are applicable only to seekable streams. Notice that both *Length* and *Position* are *long* integers, and in theory allow file sizes up to 9×10^{12}, or 9 terabytes, which should be a sufficient maximum file size for at least a couple years.

Setting the *Position* property is a straightforward way of seeking in the file. (I'll discuss a more conventional *Seek* method shortly.) For example, if *fs* is an object of type *FileStream*, you can seek to the 100th byte in the file with the statement

```
fs.Position = 100;
```

You can seek to the end of a file (for appending to the file) with the statement

```
fs.Position = fs.Length;
```

All the following methods implemented by *Stream* are overridden by *FileStream*:

***Stream* Methods (selection)**

```
int ReadByte()
int Read(byte[] abyBuffer, int iBufferOffset, int iCount)
void WriteByte(byte byValue)
void Write(byte[] abyBuffer, int iBufferOffset, int iCount)
long Seek(long lOffset, SeekOrigin so)
void SetLength(long lSize);
void Flush()
void Close()
```

You can read either individual bytes with *ReadByte* or multiple bytes with *Read*. Both methods return an *int* value, but that value means different things to each of the methods. *ReadByte* normally returns the next byte from the file cast to an *int* without sign extension. For example, the byte 0xFF becomes the integer 0x000000FF, or 255. A return value of −1 indicates an attempt to read past the end of the file.

Read returns the number of bytes read into the buffer, up to *iCount*. For files, *Read* returns the same value as the *iCount* argument unless *iCount* is greater than the remaining number of bytes in the file. A return value of 0 indicates that there are no more bytes to be read in the file. For other types of streams (network streams, for example), *Read* can return a value less than *iCount* but always at least 1 unless the entire stream has been read. The second argument to *Read* and *Write* is an offset into the buffer, not an offset into the stream!

The *Seek* method is similar to the file-seeking functions in C. The *SeekOrigin* enumeration defines where the *lOffset* argument to the *Seek* method is measured from:

***SeekOrigin* Enumeration**

Member	**Value**
Begin	0
Current	1
End	2

If the stream is writable and seekable, the *SetLength* method sets a new length for the file, possibly truncating the contents if the new length is shorter than the existing length. *Flush* causes all data in memory buffers to be written to the file.

Despite what may or may not happen as a result of garbage collection on the *FileStream* object, you should always explicitly call the *Close* method for any files you open.

If you ignore exception handling, in most cases, you can read an entire file into memory—including allocating a memory buffer based on the size of the file—in just four statements:

```
FileStream fs = new FileStream("MyFile", FileMode.Open,
                               FileAccess.Read, FileShare.Read);
Byte[] abyBuffer = new Byte[fs.Length];
fs.Read(abyBuffer, 0, (int) fs.Length);
fs.Close();
```

I say "in most cases" because this code assumes the file is less than 2^{31} bytes (or 2 gigabytes). That assumption comes into play in the casting of the last argument of the *Read* method from a 64-bit *long* to a 32-bit *int*. If the file is larger than 2 gigabytes, you'll have to read it in multiple calls to *Read*. (But you probably shouldn't even be *trying* to read a multigigabyte file entirely into memory!)

FileStream is an excellent choice for a traditional hex-dump program.

HexDump.cs

```
//--------------------------------------
// HexDump.cs © 2001 by Charles Petzold
//--------------------------------------
using System;
using System.IO;

class HexDump
{
     public static int Main(string[] astrArgs)
     {
          if (astrArgs.Length == 0)
          {
               Console.WriteLine("Syntax: HexDump file1 file2 ...");
               return 1;
          }
          foreach (string strFileName in astrArgs)
               DumpFile(strFileName);

          return 0;
     }
     protected static void DumpFile(string strFileName)
     {
          FileStream fs;

          try
          {
               fs = new FileStream(strFileName, FileMode.Open,
                                   FileAccess.Read, FileShare.Read);
          }
          catch (Exception exc)
          {
               Console.WriteLine("HexDump: {0}", exc.Message);
               return;
          }
          Console.WriteLine(strFileName);
          DumpStream(fs);
          fs.Close();
     }
     protected static void DumpStream(Stream stream)
     {
          byte[] abyBuffer = new byte[16];
          long   lAddress  = 0;
          int    iCount;

          while ((iCount = stream.Read(abyBuffer, 0, 16)) > 0)
          {
               Console.WriteLine(ComposeLine(lAddress, abyBuffer,
iCount));
               lAddress += 16;
          }
     }
     public static string ComposeLine(long lAddress, byte[] abyBuffer,
                                      int iCount)
     {
          string str = String.Format("{0:X4}-{1:X4}  ",
                         (uint) lAddress / 65536, (ushort) lAddress);

          for (int i = 0; i < 16; i++)
          {
               str += (i < iCount) ?
                              String.Format("{0:X2}", abyBuffer[i]) : "
";
               str += (i == 7 && iCount > 7) ? "-" : " ";
          }
          str += " ";

          for (int i = 0; i < 16; i++)
          {
                char ch = (i < iCount) ? Convert.ToChar(abyBuffer[i]) : '
';
                str += Char.IsControl(ch) ? "." : ch.ToString();
            }
            return str;
        }
}
```

This program uses the version of *Main* that has a single argument. The argument is an array of strings, each of which is a command-line argument to the program. Unlike the *main* function in C, the *Main* method in C# doesn't include an argument count and also doesn't include the program name among the arguments. If you run the program like so:

```
HexDump file1.cs file2.exe
```

then the argument to *Main* is a string array with two elements. Any wildcards in the arguments are *not* automatically expanded. (I'll get to wildcard expansion later in this appendix.)

Once HexDump successfully opens each file, the program uses the *Read* method to read 16-byte chunks from the file, and then HexDump's *ComposeLine* method displays them. I've reused the *ComposeLine* method in the HeadDump program in Chapter 16.

FileStream has a couple more features I want to mention briefly. For file sharing, you can lock and unlock sections of the file for exclusive use:

***FileStream* Methods (selection)**

```
void Lock(long lPosition, long lLength)
void Unlock(long lPosition, long lLength)
```

If the file system supports asynchronous reading and writing, and if you use the last constructor in the table shown earlier with a last argument of *true*, you can use the *BeginRead*, *EndRead*, *BeginWrite*, and *EndWrite* methods to read from and write to the file asynchronously.

The Problem with *FileStream*

I asserted earlier that *FileStream* is the most essential class in *System.IO* because it opens files and lets you read and write bytes. What could be more basic and vital than that?

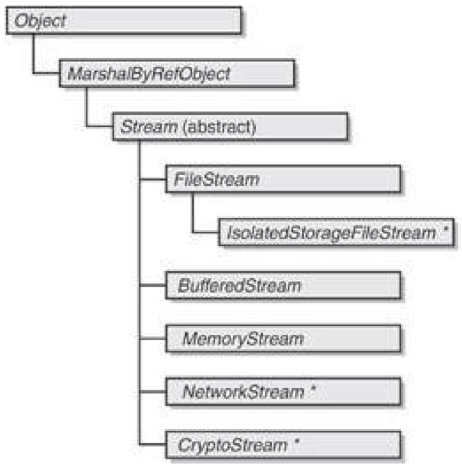
The problem, however, is that C# is not nearly as flexible as C in casting. For example, a C programmer might read an *int* from a file by taking the address of an integer variable and casting it to a byte pointer for the *fread* function. But casting something else to a byte array won't work in C#. The *Read* and *Write* methods in *FileStream* work with byte arrays and nothing but byte arrays.

Of course, because the byte is the lowest common denominator, you can always read bytes and assemble them into other basic data types (such as *char* or *int*), and you can disassemble basic types into bytes in preparation for writing. Would you like to do this yourself? I didn't think so.

So, unless reading and writing arrays of bytes is entirely satisfactory to you, you probably can't limit your knowledge of file I/O to the *FileStream* class. As I'll explain shortly, you use the *StreamReader* and *StreamWriter* classes for reading and writing text files, and *BinaryReader* and *BinaryWriter* for reading and writing binary files of types other than byte arrays.

Other Stream Classes

The *FileStream* class is one of several classes descended from the abstract class *Stream*. For a class that can't be instantiated, *Stream* plays a very important role in the .NET Framework. This hierarchy diagram shows six classes descended from *Stream*:



The stream classes with an asterisk are defined in namespaces other than *System.IO*.

In addition, a number of methods in other classes scattered throughout the .NET Framework return objects of type *Stream*. For example, as I'll demonstrate later in this appendix, a .NET program that reads files from the Web does so using a *Stream* object. A program in Chapter 11 demonstrates that you can also load image files (such as JPEGs) from streams.

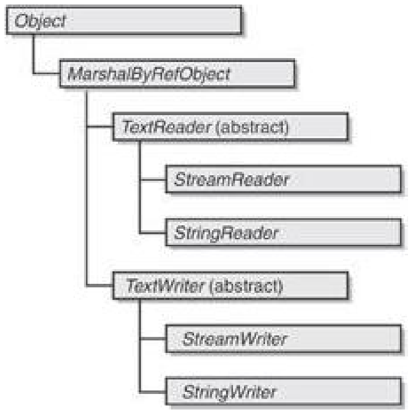
For performance purposes, the *FileStream* class creates a buffered stream. An area of memory is maintained so that every call to *ReadByte*, *Read*, *WriteByte*, and *Write* doesn't necessarily result in a call to the operating system to read from or write to the file.

If you have a *Stream* object that isn't a buffered stream, you can convert it to a buffered stream using the *BufferedStream* class.

The *MemoryStream* class lets you create an expandable area of memory that you can access using the *Stream* methods. I demonstrate how to use the *MemoryStream* class in the CreateMetafileMemory program in Chapter 23 and in several programs in Chapter 24.

Reading and Writing Text

One important type of file is the text file, which consists entirely of lines of text separated by end-of-line markers. The *System.IO* class has specific classes to read and write text files. Here's the object hierarchy:



Although these classes are not descended from *Stream*, they almost certainly make use of the *Stream* class.

The two classes I'm going to focus on here are *StreamReader* and *StreamWriter*, which are designed for reading and writing text files or text streams. The two other nonabstract classes are *StringReader* and *StringWriter*, which are not strictly file I/O classes. They use similar methods to read to and write from strings. I discuss these classes briefly at the end of Appendix C and demonstrate the *StringWriter* class in the EnumMetafile program in Chapter 23.

Text may seem to be a very simple form of data storage, but in recent years, text has assumed a layer of complexity as a result of the increased use of Unicode.

The *System.Char* data type in .NET—and the *char* alias in C#—is a 16-bit value representing a character in the Unicode character set. The .NET *System.String* type (and the C# *string* alias) represents a string of Unicode characters. But what happens when you write strings from a C# program to a file? Do you want to write them as Unicode? That makes sense only if every application that reads the file you create expects to be reading Unicode! You probably want to avoid Unicode if you know that other applications reading the file are anticipating encountering 8-bit ASCII characters.

The first 256 characters in Unicode are the same as the 128 characters of ASCII and the 128 characters of the ISO Latin Alphabet No. 1 extension to ASCII. (The combination of these two character sets is often referred to in Windows API documentation as the *ANSI* character set.) For example, the capital *A* is 0x41 in ASCII and 0x0041 in Unicode. Unicode strings that contain exclusively (or mostly) ASCII contain a lot of zeros. These zeros cause problems for a lot of traditional C-based and UNIX-based programs because those programs interpret a zero byte as a string-termination character.

To alleviate these problems, the *StreamWriter* class lets you have control over how the Unicode strings in your C# program are converted for storage in a file. You assert this control via classes defined in the *System.Text* namespace. Similarly, *StreamReader* lets your program read text files in various formats and convert the text from the files to Unicode strings in your program.

Let's look at *StreamWriter* first. You use this class to write to new or existing text files.

Four of the *StreamWriter* constructors let you create an object of type *StreamWriter* based on a filename:

StreamWriter Constructors (selection)

```
StreamWriter(string strFileName)
StreamWriter(string strFileName, bool bAppend)
StreamWriter(string strFileName, bool bAppend, Encoding enc)
StreamWriter(string strFileName, bool bAppend, Encoding enc, int
iBufferSize)
```

These constructors open the file for writing, probably using a *FileStream* constructor internally. By default, if the file exists, the contents will be destroyed. The *bAppend* argument allows you to override that default action. The remaining constructors create an object of type *StreamWriter* based on an existing *Stream* object:

StreamWriter Constructors (selection)

```
StreamWriter(Stream stream)
StreamWriter(Stream stream, Encoding enc)
StreamWriter(Stream stream, Encoding enc, int iBufferSize)
```

If you use a constructor without an *Encoding* argument, the resultant *StreamWriter* object will *not* store strings to the file in a Unicode format with 2 bytes per character. Nor will it convert your strings to ASCII. Instead, the *StreamWriter* object will store strings in a format known as UTF-8, which is something I'll go over shortly.

If you use one of the *StreamWriter* constructors with an *Encoding* argument, you need an object of type *Encoding*, which is a class defined in the *System.Text* namespace. It's easiest (and in many cases, sufficient) to use one of the static properties of the *Encoding* class to obtain this object:

***Encoding* Static Properties**

Type	Property	Accessibility
Encoding	*Default*	get
Encoding	*Unicode*	get
Encoding	*BigEndianUnicode*	get
Encoding	*UTF8*	get
Encoding	*UTF7*	get
Encoding	*ASCII*	get

The *Encoding* argument to the *StreamWriter* constructor can also be an instance of one of the classes in *System.Text* that derive from *Encoding*, which are *ASCIIEncoding*, *UnicodeEncoding*, *UTF7Encoding*, and *UTF8Encoding*. The constructors for these classes often have a few options, so you may want to check them out if the static properties aren't doing precisely what you want.

When you specify an encoding of *Encoding.Unicode*, each character is written to the file in 2 bytes with the least significant byte first, in accordance with the so-called little-endian architecture of Intel microprocessors. The file or stream begins with the bytes 0xFF and 0xFE, which correspond to the Unicode character 0xFEFF, which is defined in the Unicode standard as the byte order mark (BOM).

An encoding of *Encoding.BigEndianUnicode* stores the most significant byte of each character first. The file or stream begins with the bytes 0xFE and 0xFF, which also correspond to the Unicode character 0xFEFF. The Unicode character 0xFFFE is intentionally undefined so that applications can determine the byte ordering of a Unicode file from its first two bytes.

If you want to store strings in Unicode but you don't want the byte order marks emitted, you can instead obtain an *Encoding* argument for the *StreamWriter* constructor by creating an object of type *UnicodeEncoding*:

```
new UnicodeEncoding(bBigEndian, bIncludeByteOrderMark)
```

Set the two Boolean arguments appropriately.

UTF-8 is a character encoding designed to represent Unicode characters without using any zero bytes (and hence, to be C and UNIX friendly). UTF stands for *UCS Transformation Format*. UCS stands for *Universal Character Set*, which is another name for ISO 10646, a character-encoding standard with which Unicode is compatible.

In UTF-8, each Unicode character is translated to a sequence of 1 to 6 nonzero bytes. Unicode characters in the ASCII range (0x0000 through 0x007F) are translated directly to single-byte values. Thus, Unicode strings that contain only ASCII are translated to ASCII files. UTF-8 is documented in RFC 2279. (RFC stands for Request for Comments. RFCs are documentations of Internet standards. You can obtain RFCs from many sources, including the Web site of the Internet Engineering Task Force, *http://www.ietf.org*.)

When you specify *Encoding.UTF8*, the *StreamWriter* class converts the Unicode text strings to UTF-8. In addition, it writes the three bytes 0xEF, 0xBB, and 0xBF to the beginning of the file or stream. These bytes are the Unicode BOM converted to UTF-8.

If you want to use UTF-8 encoding but you don't want those three bytes emitted, don't use *Encoding.UTF8*. Use *Encoding.Default* instead or one of the constructors that doesn't have an *Encoding* argument. These options also provide UTF-8 encoding, but the three identification bytes are not emitted.

Alternatively, you can create an object of type *UTF8Encoding* and pass that object as the argument to *StreamWriter*. Use

```
new UTF8Encoding()
```

or

```
new UTF8Encoding(false)
```

to suppress the three bytes, and use

```
new UTF8Encoding(true)
```

to emit the identification bytes.

UTF-7 is documented in RFC 2152. Unicode characters are translated to a sequence of bytes that has an upper bit of 0. UTF-7 is intended for environments in which only 7-bit values can be used, such as e-mail. Use *Encoding.UTF7* in the *StreamWriter* constructor for UTF-7 encoding. No identification bytes are involved with UTF-7.

When you specify an encoding of *Encoding.ASCII*, the resultant file or stream contains only ASCII characters, that is, characters in the range 0x00 through 0x7F. Any Unicode character not in this range is converted to a question mark (ASCII code 0x3F). This is the only encoding in which data is actually lost.

The *StreamWriter* class has a few handy properties:

***StreamWriter* Properties (selection)**

Type	Property	Accessibility
Stream	*BaseStream*	get
Encoding	*Encoding*	get
bool	*AutoFlush*	get/set
string	*NewLine*	get/set

The *BaseStream* property returns either the *Stream* object you used to create the *StreamWriter* object or the *Stream* object that the *StreamWriter* class created based on the filename you supplied. If the base stream supports seeking, you can use that object to perform seeking operations on that stream.

The *Encoding* property returns the encoding you specified in the constructor or *UTF8Encoding* if you specified no encoding. Setting *AutoFlush* to *true* performs a flush of the buffer after every write.

The *NewLine* property is inherited from *TextWriter*. By default, it's the string "\r\n" (carriage return and line feed), but you can change it to "\n" (line feed). If you change it to anything else, the files won't be readable by *StreamReader* objects.

The versatility of the *StreamWriter* class involves the *Write* and *WriteLine* methods that the class inherits from *TextWriter*:

***TextWriter* Methods (selection)**

```
void Write(...)
void WriteLine(...)
void Flush()
void Close()
```

TextWriter supports (and *StreamWriter* inherits) 17 versions of *Write* and 18 versions of *WriteLine* that let you specify any object as an argument to the method. The object you specify is converted to a string by the use of its *ToString* method. The *WriteLine* method follows the string with an end-of-line marker. A version of *WriteLine* with no arguments writes just an end-of-line marker. The *Write* and *WriteLine* methods also include versions with formatting strings, just as the *Console.Write* and *Console.WriteLine* methods do.

Here's a tiny program that appends text to the same file every time you run the program.

StreamWriterDemo.cs

```
//-----------------------------------------------
// StreamWriterDemo.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.IO;

class StreamWriterDemo
{
     public static void Main()
     {
          StreamWriter sw = new StreamWriter("StreamWriterDemo.txt",
true);

          sw.WriteLine("You ran the StreamWriterDemo program on {0}",
                       DateTime.Now);

          sw.Close();
     }
}
```

Notice the *true* argument to the constructor, indicating that the file will be appended to. The Unicode strings in the program are converted to UTF-8, but they will appear to be ASCII.

I mentioned the *Console* class a moment ago. The input and output devices in that class are defined as objects of type *TextWriter*. Try inserting the following lines at the beginning of the *Main* method in HexDump:

```
StreamWriter sw = new StreamWriter("prn", false, Encoding.ASCII);
Console.SetOut(sw);
```

You'll also need to add a *using* statement:

```
using System.Text;
```

Now all the output from the program goes to the printer.

The *StreamReader* class is for reading text files or streams. There are five constructors for opening a text file for reading:

StreamReader Constructors (selection)

```
StreamReader(string strFileName)
StreamReader(string strFileName, Encoding enc)
StreamReader(string strFileName, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect, int
iBufferSize)
```

There is an additional set of five constructors for creating a *StreamReader* object based on an existing stream:

StreamReader Constructors (selection)

```
StreamReader(Stream stream)
StreamReader(Stream stream, Encoding enc)
StreamReader(Stream stream, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect, int iBufferSize)
```

If you set *bDetect* to *true*, the constructor will attempt to determine the encoding of the file from the first two or three bytes. Or you can specify the encoding explicitly. If you set *bDetect* to *true* and also specify an encoding, the constructor will use the specified encoding only if it can't detect the encoding of the file. (For example, ASCII and UTF-7 can't be differentiated by inspection because they don't begin with a BOM and both contain only bytes in the range 0x00 through 0x7F.)

The *StreamReader* class has the following two, read-only properties:

StreamReader Properties

Type	Property	Accessibility
Stream	*BaseStream*	get
Encoding	*CurrentEncoding*	get

The *CurrentEncoding* property may change between the time the object is constructed and the first read operation performed on the file or stream because the object has knowledge of identification bytes only after the first read.

Here are the methods to peek, read, and close text files:

StreamReader Methods (selection)

```
int Peek()
int Read()
int Read(char[] achBuffer, int iBufferOffset, int iCount)
string ReadLine()
string ReadToEnd()
void Close()
```

The *Peek* and the first *Read* methods both return the next character in the stream or –1 if the end of the stream has been reached. You must explicitly cast the return value to a *char* if the return value is not –1. The second *Read* method returns the number of characters read or 0 if the end of the stream has been reached.

The *ReadLine* method reads the next line up to the next end-of-line marker and strips the end-of-line characters from the resultant string. The method returns a zero-length character string if the line of text contains only an end-of-line marker; the method returns *null* if the end of the stream has been reached.

ReadToEnd returns everything from the current position to the end of the file. The method returns *null* if the end of the stream has been reached.

Here's a program that assumes the command-line argument is a URI (Universal Resource Identifier) of an HTML file (or other text file) on the Web. It obtains a *Stream* for that file using some boilerplate code involving the *WebRequest* and *WebResponse* classes. It then constructs a *StreamReader* object from that stream, uses *ReadLine* to read each line, and then displays each line using *Console.WriteLine* with a line number.

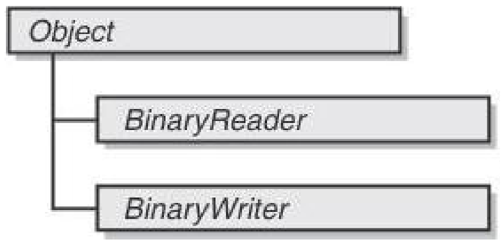
HtmlDump.cs

```
//---------------------------------------
// HtmlDump.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
using System.IO;
using System.Net;

class HtmlDump
{
     public static int Main(string[] astrArgs)
     {
          if (astrArgs.Length == 0)
          {
               Console.WriteLine("Syntax: HtmlDump URI");
               return 1;
          }

          WebRequest webreq;
          WebResponse webres;

          try
          {
               webreq = WebRequest.Create(astrArgs[0]);
               webres = webreq.GetResponse();
          }
          catch (Exception exc)
          {
               Console.WriteLine("HtmlDump: {0}", exc.Message);
               return 1;
          }

          if (webres.ContentType.Substring(0, 4) != "text")
          {
               Console.WriteLine("HtmlDump: URI must be a text type.");
               return 1;
          }

          Stream       stream = webres.GetResponseStream();
          StreamReader strrdr = new StreamReader(stream);
          string       strLine;
          int          iLine = 1;

          while ((strLine = strrdr.ReadLine()) != null)
```

```
                Console.WriteLine("{0:D5}: {1}", iLine++, strLine);

            stream.Close();
            return 0;
        }
}
```

Binary File I/O

By definition, any file that's not a text file is a binary file. I've already discussed the *FileStream* class, which lets you read and write bytes. But most binary files consist of data types that are stored as multiple bytes. Unless you want to write code that constructs and deconstructs integers and other types from their constituent bytes, you'll want to take advantage of the *BinaryReader* and *BinaryWriter* classes, both of which are derived solely from *Object*:



The constructors for these classes require a *Stream* object. If you want to use a file with these classes, create a new *FileStream* object (or obtain one from some other means) first. For the *BinaryWriter* class, the *Encoding* you optionally specify affects the storage of text in the stream:

BinaryWriter Constructors

```
BinaryWriter(Stream stream)
BinaryWriter(Stream stream, Encoding enc)
```

The constructors for *BinaryReader* are identical:

BinaryReader Constructors

```
BinaryReader(Stream stream)
BinaryReader(Stream stream, Encoding enc)
```

Both classes have a single read-only property named *BaseStream* that is the *Stream* object you specified in the constructor.

The *Write* methods in *BinaryWriter* are defined for all the basic types as well as for arrays of bytes and characters.

BinaryWriter Public Methods

```
void Write(...)
void Write(byte[] abyBuffer, int iBufferOffset, int iBytesToWrite)
void Write(char[] achBuffer, int iBufferOffset, int iBytesToWrite)
long Seek(int iOffset, SeekOrigin so)
void Flush()
void Close()
```

You can use an object of any basic type (*bool*, *byte*, *sbyte*, *byte[]*, *char*, *char[]*, *string*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, or *decimal*) as an argument to *Write*.

These methods do not store any information about the type of the data. Each type uses as many bytes as necessary. For example, a *float* is stored in 4 bytes. A *bool* requires 1 byte. The sizes of arrays are not stored. A 256-element byte array is stored in 256 bytes.

Strings stored in the file are preceded by the byte length stored as a 7-bit encoded integer. (The 7-bit integer encoding uses as many bytes as necessary to store an integer in 7-bit chunks. The first byte of storage is the lowest 7 bits of the integer, and so forth. The high bit of each byte is 1 if there are more bytes. The *BinaryWriter* class includes a protected method named *Write7BitEncodedInt* that performs this encoding.)

The *Close* method closes the underlying stream that the *BinaryWriter* object is based on.

The *BinaryReader* class has separate methods to read all the various types.

***BinaryReader* Methods (selection)**

```
bool ReadBoolean()
byte ReadByte()
byte[] ReadBytes(int iCount)
sbyte ReadSByte()
char ReadChar()
char[] ReadChars(int iCount)
short ReadInt16()
int ReadInt32()
long ReadInt64()
ushort ReadUInt16()
uint ReadUInt32()
ulong ReadUInt64()
float ReadSingle()
double ReadDouble()
decimal ReadDecimal()
```

These methods throw an exception of type *EndOfStreamException* if the end of the stream has been reached. In most cases, your program will know the format of a binary file it's accessing and can avoid end-of-stream conditions. However, for maximum protection, you should put your read statements in *try* blocks in case you encounter corrupted files.

You can also read individual characters, or arrays of bytes or characters:

***BinaryReader* Methods (selection)**

```
int PeekChar()
int Read()
void Read(byte[] abyBuffer, int iBufferOffset, int iBytesToRead)
void Read(char[] achBuffer, int iBufferOffset, int iBytesToRead)
void Close()
```

The *PeekChar* and *Read* methods involve characters, not bytes, and will assume that the file is UTF-8 encoded if you don't explicitly indicate an encoding in the constructor. The methods return −1 if the end of the stream has been reached.

If you have experience with file I/O in C programs, you're probably familiar with common techniques to read and write data structures in a binary format. For example, you may define a structure like so:

```
typedef struct
{
     int   i;
     float f;
     char  ch[10];
     int   j;
     float g;
}
STRUCTDEF;
```

and a variable of type *STRUCTDEF* like this:

```
STRUCTDEF mystruct;
```

If you open a file with *fopen* and name the *FILE* pointer *file*, you can then write out the contents of the structure using *fwrite*, as here,

```
fwrite(&mystruct, sizeof(STRUCTDEF), 1, file);
```

and read it back in similarly:

```
fread(&mystruct, sizeof(STRUCTDEF), 1, file);
```

This job is so easy because C stores the contents of a structure as just a block of memory. The first argument of the *fwrite* and *fread* functions is defined as a *void* pointer, so you can specify a pointer to anything.

With C#, you don't have quite this much casting freedom. You'll probably want to take a completely different (and more structured) approach to reading and writing binary data. Instead of defining structures such as *STRUCTDEF*, you'll be defining classes. When you save an instance of a class to a file, you want to save sufficient information to re-create that object when you read the file. In a well-designed C# class, you'll probably be saving all the properties of the class that are necessary to re-create the object.

Let's assume you have a class named *SampleClass* that has three properties necessary to re-create the object: a *float* named *Value*, a *string* named *Text*, and an object of type *Fish* stored as a property named *BasicFish*. (*Fish* is another class you've created.) *SampleClass* also has a constructor defined to create a new object from these three items:

```
public SampleClass(float fValue, string strText, Fish fish)
```

Let's also assume that you need to use a binary file to store information that consists of many objects, including objects of type *SampleClass*. Each class you create can implement both an instance method named *Write* and a static method named *Read*. Here's the *Write* method for *SampleClass*. Notice the *BinaryWriter* argument.

```
public void Write(BinaryWriter bw)
{
     bw.Write(Value);
     bw.Write(Text);
     BasicFish.Write(bw);
}
```

Because the *Value* and *Text* properties are basic types, this method can simply call the *Write* method of *BinaryWriter* for them. But for the *BasicFish* property, it must call the similar *Write* method you've also implemented in the *Fish* class, passing to it the *BinaryWriter* argument.

The *Read* method is static because it must create an instance of *SampleClass* after reading binary data from the file:

```
public static SampleClass Read(BinaryReader br)
{
     float fValue = br.ReadSingle();
     string strText = br.ReadString();
     Fish fish = Fish.Read(br);
     return new SampleClass(fValue, strText, fish);
}
```

Notice that the *Fish* class must also have a similar static *Read* method.

The *Environment* Class

Let's leave the *System.IO* namespace briefly to take a look at the *Environment* class, which is defined in the *System* namespace. *Environment* has a collection of miscellaneous properties and methods that are useful for obtaining information about the machine on which the program is running and the current user logged on to the machine. As its name suggests, the *Environment* class also allows a program to obtain environment strings. (I make use of this latter facility in the EnvironmentVars program in Chapter 18.)

Two methods in *Environment* provide information about the file system:

***Environment* Static Methods (selection)**

```
string[] GetLogicalDrives()
string GetFolderPath(Environment.SpecialFolder sf)
```

I have a fairly normal system with a CD-ROM drive and an Iomega Zip drive, so on my machine, *GetLogicalDrives* returns the following four strings, in this order:

```
A:\
C:\
D:\
E:\
```

The argument to *GetFolderPath* is a member of the *Environment.SpecialFolder* enumeration. The rightmost column in the following table indicates the return string from *GetFolderPath* on a machine running the default installation of Windows 2000, where I've used an ellipsis to indicate that the return string includes the user's name (which is the same as the value returned from the static property *Environment.UserName*).

***Environment.SpecialFolder* Enumeration**

Member	Value	Common Return Values
Programs	2	C:\Documents and Settings\...\Start Menu\Programs
Personal	5	C:\Documents and Settings\...\My Documents
Favorites	6	C:\Documents and Settings\...\Favorites
Startup	7	C:\Documents and Settings\...\Start Menu\Programs\Startup
Recent	8	C:\Documents and Settings\...\Recent

***Environment.SpecialFolder* Enumeration**

Member	**Value**	**Common Return Values**
SendTo	9	C:\Documents and Settings\...\SendTo
StartMenu	11	C:\Documents and Settings\...\Start Menu
DesktopDirectory	16	C:\Documents and Settings\...\Desktop
Templates	21	C:\Documents and Settings\...\Templates
ApplicationData	26	C:\Documents and Settings\...\Application Data
LocalApplicationData	28	C:\Documents and Settings\...\Local Settings\Application Data
InternetCache	32	C:\Documents and Settings\...\Local Settings\Temporary Internet Files
Cookies	33	C:\Documents and Settings\...\Cookies
History	34	C:\Documents and Settings\...\Local Settings\History
CommonApplicationData	35	C:\Documents and Settings\All Users\Application Data
System	37	C:\WINNT\System32
ProgramFiles	38	C:\Program Files
CommonProgramFiles	43	C:\Program Files\Common Files

Oddly enough, the *SpecialFolder* enumeration is defined within the *Environment* class. Instead of calling *GetFolderPath* as

```
Environment.GetFolderPath(SpecialFolder.Personal)    // Won't work!
```

you need to preface *SpecialFolder* with the class in which it's defined:

```
Environment.GetFolderPath(Environment.SpecialFolder.Personal)
```

The *Environment* class also includes a couple properties that relate to the file system and file I/O:

***Environment* Static Properties (selection)**

Type	**Property**	**Accessibility**
string	*SystemDirectory*	get
string	*CurrentDirectory*	get/set

The *SystemDirectory* property returns the same string as the *GetFolderPath* method with the *Environment.SpecialFolder.System* argument.

The *CurrentDirectory* property lets a program obtain or set the current drive and directory for the application. When setting the directory, you can use a relative directory path, including the ".." string to indicate the parent directory. To change to the root directory of another drive, use the drive letter like so:

```
Environment.CurrentDirectory = "D:\\";
```

If the current drive and directory are on a drive other than C and you use

```
Environment.CurrentDirectory = "C:";
```

the current directory is set to the last current directory on drive C before the current drive was changed to something other than C. This technique doesn't seem to work with other drives. The call

```
Environment.CurrentDirectory = "D:";
```

always seems to set the current directory as the root directory of drive D.

As you'll see shortly, other classes defined in the *System.IO* namespace have equivalents to *GetLogicalDrives* and *CurrentDirectory*.

File and Path Name Parsing

At times, you need to parse and scan filenames and path names. For example, your program may have a fully qualified filename and you may need just the directory or the drive. The *Path* class, defined in the *System.IO* namespace, consists solely of static methods and static read-only fields that ease jobs like these.

In the following table, the right two columns show sample return values from the methods when the *strFileName* argument is the indicated string at the top of the column. In these examples, I'm assuming the current directory is C:\Docs.

***Path* Static Methods (examples)**

Method	\DirA\MyFile	DirA\MyFile.txt
`bool IsPathRooted(string strFileName)`	true	false
`bool HasExtension(string strFileName)`	false	true
`string GetFileName(string strFileName)`	MyFile	MyFile.txt
`string GetFileNameWithoutExtension (string strFileName)`	MyFile	MyFile
`string GetExtension(string strFileName)`		.txt
`string GetDirectoryName (string strFileName)`	\DirA	DirA
`string GetFullPath(string strFileName)`	C:\DirA\MyFile	C:\Docs\DirA\MyFile.txt
`string GetPathRoot(string strFileName)`	\	

What's interesting here is that neither DirA nor MyFile has to exist for these methods to work. The methods are basically performing string manipulation, possibly in combination with the current directory.

The following two methods return a new path and filename:

***Path* Static Methods (selection)**

```
string Combine(string strLeftPart, string strRightPart)
string ChangeExtension(string strFileName, string strNewExtension)
```

The *Combine* method joins together a path name (on the left) with a path and/or filename (on the right). Use *Combine* rather than string concatenation for this job. Otherwise, you have to worry about whether a backslash is the end of the left part or the beginning of the right part. The *ChangeExtension* method simply changes the filename extension from one string to another. Include a period in the new extension. Set the *strNewExtension* argument to *null* to remove the extension.

The following methods obtain an appropriate directory for storing temporary data and a fully qualified unique filename the program can use to store temporary data:

***Path* Static Methods (selection)**

```
string GetTempPath()
string GetTempFileName()
```

If you must do your own file and path name parsing, don't hard-code characters that you think you'll encounter in the strings. Use the following static read-only fields of *Path* instead:

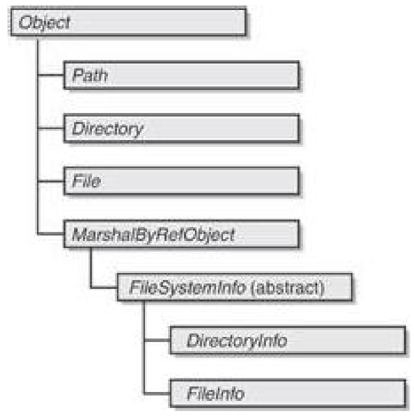
***Path* Static Fields**

Type	Field	Accessibility	Windows Default
char	*PathSeparator*	read-only	;
char	*VolumeSeparatorChar*	read-only	:
char	*DirectorySeparatorChar*	read-only	\
char	*AltDirectorySeparatorChar*	read-only	/
char[]	*InvalidPathChars*	read-only	" < > |

Parallel Classes

Another common file I/O job is obtaining a list of all files and subdirectories in a directory. Historically, this job has always been a bit awkward. The standard libraries associated with the C programming language didn't include such a facility, probably because UNIX directory lists were text files that programs could directly access and parse.

Four classes provide you with information about files and directories: *Directory*, *File*, *DirectoryInfo*, and *FileInfo*. All four of these classes (as well as the *Path* class I just described) are sealed and can't be inherited. Here's the class hierarchy:



Directory and *File* can't be instantiated; the two classes consist solely of static methods.

DirectoryInfo and *FileInfo* contain *no* static methods or properties, and you must obtain an object of type *DirectoryInfo* or *FileInfo* to use these classes. Both classes derive from the abstract class *FileSystemInfo*, so they share some properties and methods.

As the names suggest, *Directory* and *DirectoryInfo* provide similar methods, except that the *Directory* methods are static and require an argument that is a directory name. The *DirectoryInfo* properties and methods are not static; the constructor argument indicates the directory name to which the properties and methods apply.

Similarly, *File* and *FileInfo* provide corresponding methods, except that you indicate a particular filename in the static *File* method calls and you create an instance of *File* by specifying a filename in the constructor.

If you need information about a particular file, you may wonder whether it's best to use *File* or *FileInfo* (or similarly for directories, whether to use *Directory* or *DirectoryInfo*). If you need only one item of information, it's probably easiest to use the appropriate static method in *File* or *Directory*. However, if you need multiple items, it makes more sense to create an object of type *FileInfo* or *DirectoryInfo* and then use the instance properties and methods. But don't feel pressured to use one class in preference to the other.

Working with Directories

Let's begin with the *Directory* and *DirectoryInfo* classes. The following three static methods of the *Directory* class have no equivalents in the *DirectoryInfo* class:

Directory Static Methods (selection)

```
string[] GetLogicalDrives()
string GetCurrentDirectory()
void SetCurrentDirectory(string strPath)
```

These methods essentially duplicate the static *GetLogicalDrives* method and the *CurrentDirectory* property of the *Environment* class.

To use any of the properties or methods of the *DirectoryInfo* class, you need a *DirectoryInfo* object. One of the ways in which you can obtain such an object is by using the *DirectoryInfo* constructor:

DirectoryInfo Constructor

```
DirectoryInfo(string strPath)
```

The directory doesn't have to exist. Indeed, if you want to create a new directory, creating an object of type *DirectoryInfo* is a first step.

After creating an object of type *DirectoryInfo*, you can determine whether the directory exists. Even if the directory doesn't exist, you can obtain certain information about the directory as if it did exist. The two rightmost columns of the following table show examples. The column heading is the string passed to the *DirectoryInfo* constructor. The current directory is assumed to be C:\Docs.

DirectoryInfo Properties (selection)

Type	Property	Accessibility	DirA	DirA\DirB.txt
bool	*Exists*	get		
string	*Name*	get	DirA	DirB.txt
string	*FullName*	get	C:\Docs\DirA	C:\Docs\DirA\DirB.txt
string	*Extension*	get		.txt
DirectoryInfo	*Parent*	get	C:\Docs	C:\Docs\DirA
DirectoryInfo	*Root*	get	C:\	C:\

FullName and *Extension* are inherited from the *FileSystemInfo* class.

A few of these properties are also duplicated as static methods in the *Directory* class. Because they are static methods, they require an argument indicating the path name you're interested in:

Directory Static Methods (selection)

```
bool Exists(string strPath)
DirectoryInfo GetParent(string strPath)
```

```
string GetDirectoryRoot(string strPath)
```

I mentioned earlier that you can create a *DirectoryInfo* object based on a directory that doesn't exist. You can then create that directory on the disk by calling the *Create* method, or you can create a subdirectory of the directory:

DirectoryInfo Methods (selection)

```
void Create()
DirectoryInfo CreateSubdirectory(string strPath)
void Refresh()
```

Notice that the *CreateSubdirectory* call returns another *DirectoryInfo* object with information about the new directory. If the indicated directory already exists, *no* exception is thrown. The directory used to create the *DirectoryInfo* object or passed to *CreateSubdirectory* can contain multiple levels of directory names.

If the directory doesn't exist when you create the *DirectoryInfo* object and you then call *Create*, the *Exists* property won't suddenly become *true*. You must call the *Refresh* method (inherited from *FileSystemInfo*) to refresh the *DirectoryInfo* information.

The *Directory* class also has a static method to create a new directory:

Directory Static Methods (selection)

```
DirectoryInfo CreateDirectory(string strPath)
```

You can delete directories using the *Delete* method of *DirectoryInfo*:

DirectoryInfo Delete Methods

```
void Delete()
void Delete(bool bRecursive)
```

The methods have corresponding static versions in the *Directory* class:

Directory Delete Static Methods

```
void Delete(string strPath)
void Delete(string strPath, bool bRecursive)
```

If you use the second version of *Delete* in either table and you set the *bRecursive* argument to *true*, the method also erases all files and subdirectories in the indicated directory. Otherwise, the directory must be empty or an exception will be thrown.

Although the following information is more useful in connection with files, this table of four properties completes our survey of the *DirectoryInfo* properties:

DirectoryInfo Properties (selection)

Type	**Property**	**Accessibility**
FileAttributes	*Attributes*	get/set
DateTime	*CreationTime*	get/set
DateTime	*LastAccessTime*	get/set
DateTime	*LastWriteTime*	get/set

These properties are all inherited from the *FileSystemInfo* class, and except for *Attributes*, they are all duplicated by static methods in the *Directory* class:

Directory Static Methods (selection)

```
DateTime GetCreationTime(string strPath)
DateTime GetLastAccessTime(string strPath)
DateTime GetLastWriteTime(string strPath)
void SetCreationTime(string strPath, DateTime dt)
void SetLastAccessTime(string strPath, DateTime dt)
void SetLastWriteTime(string strPath, DateTime dt)
```

The *DateTime* structure is defined in the *System* namespace. *FileAttributes* is a collection of bit flags defined as an enumeration:

FileAttributes Enumeration

Member	**Value**
ReadOnly	0x00000001
Hidden	0x00000002
System	0x00000004
Directory	0x00000010
Archive	0x00000020
Device	0x00000040
Normal	0x00000080
Temporary	0x00000100
SparseFile	0x00000200
ReparsePoint	0x00000400
Compressed	0x00000800
Offline	0x00001000
NotContentIndexed	0x00002000
Encrypted	0x00004000

Directories always have the *Directory* bit (0x10) set.

To move a directory and all its contents to another location on the same disk, you can use the *MoveTo* method:

DirectoryInfo Methods (selection)

```
void MoveTo(string strPathDestination)
```

Or you can use the static *Move* method in the *Directory* class:

Directory Static Methods (selection)

```
void Move(string strPathSource, string strPathDestination)
```

With either method call, the destination must not currently exist.

The remaining methods of *DirectoryInfo* and *Directory* obtain an array of all the files and subdirectories in a directory, or only those directories and files that match a specified pattern using wildcards (question marks and asterisks). Here are the six methods of *DirectoryInfo*:

DirectoryInfo Methods (selection)

```
DirectoryInfo[] GetDirectories()
DirectoryInfo[] GetDirectories(string strPattern)
FileInfo[] GetFiles()
FileInfo[] GetFiles(string strPattern)
FileSystemInfo[] GetFileSystemInfos()
FileSystemInfo[] GetFileSystemInfos(string strPattern)
```

The *GetDirectories* method returns a collection of directories as an array of *DirectoryInfo* objects. Likewise, the *GetFiles* method returns a collection of files as an array of *FileInfo* objects. The *GetFileSystemInfos* method returns both directories and files as an array of *FileSystemInfo* objects. You'll recall that *FileSystemInfo* is the parent class for both *DirectoryInfo* and *FileInfo*.

The *Directory* class has a similar set of six methods, but these all return arrays of strings:

Directory Static Methods (selections)

```
string[] GetDirectories(string strPath)
string[] GetDirectories(string strPath, string strPattern)
string[] GetFiles(string strPath)
string[] GetFiles(string strPath, string strPattern)
string[] GetFileSystemEntries(string strPath)
string[] GetFileSystemEntries(string strPath, string strPattern)
```

We're now fully equipped to enhance the HexDump program shown earlier so that it works with wildcard file specifications on the command line. Here's WildCardHexDump.

WildCardHexDump.cs

```
//---------------------------------------------
// WildCardHexDump.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.IO;
```

```
class WildCardHexDump: HexDump
{
     public new static int Main(string[] astrArgs)
     {
          if (astrArgs.Length == 0)
          {
               Console.WriteLine("Syntax: WildCardHexDump file1 file2
...");
               return 1;
          }
          foreach (string str in astrArgs)
               ExpandWildCard(str);

          return 0;
     }
     static void ExpandWildCard(string strWildCard)
     {
          string[] astrFiles;

          try
          {
               astrFiles = Directory.GetFiles(strWildCard);
          }
          catch
          {
               try
               {
                    string strDir  = Path.GetDirectoryName(strWildCard);
                    string strFile = Path.GetFileName(strWildCard);

                    if (strDir == null || strDir.Length == 0)
                         strDir = ".";

                    astrFiles = Directory.GetFiles(strDir, strFile);
               }
               catch
               {
                    Console.WriteLine(strWildCard + ": No Files found!");
                    return;
               }
          }
          if (astrFiles.Length == 0)
               Console.WriteLine(strWildCard + ": No files found!");

          foreach(string strFile in astrFiles)
```

```
                DumpFile(strFile);
        }
}
```

Besides normal wildcards, I wanted to be able to specify just a directory name as an argument. For example, I wanted

```
WildCardHexDump c:\
```

to be the equivalent of

```
WildCardHexDump c:\*.*
```

The *ExpandWildCard* method begins by attempting to obtain all the files in the particular command-line argument:

```
astrFiles = Directory.GetFiles(strWildCard);
```

This call will work if *strWildCard* specifies only a directory (such as "c:\"). Otherwise, it throws an exception. That's why it's in a *try* block. The *catch* block assumes that the command-line argument has path and filename components, and it obtains these components using the static *GetDirectoryName* and *GetFileName* methods of *Path*. However, the *GetFiles* method of *Directory* doesn't want a first argument that is *null* or an empty string. Before calling *GetFiles*, the program avoids that problem by setting the path name to ".", which indicates the current directory.

File Manipulation and Information

Like the *Directory* and *DirectoryInfo* classes, the *File* and *FileInfo* classes are very similar and share many properties and methods. Like the *Directory* class, all the methods in the *File* class are static, and the first argument to every method is a string that indicates the path name of the file. The *FileInfo* class inherits from *FileSystemInfo*. You create an object of type *FileInfo* based on a filename that could include a full or a relative directory path.

FileInfo Constructor

```
FileInfo(string strFileName)
```

The file doesn't have to exist. You can determine whether the file exists, and also some information about it, with the following read-only properties:

***FileInfo* Properties (selection)**

Type	Property	Accessibility
bool	*Exists*	get
string	*Name*	get
string	*FullName*	get
string	*Extension*	get
string	*DirectoryName*	get
DirectoryInfo	*Directory*	get
long	*Length*	get

Only one of these properties is duplicated in the *File* class:

File Methods

```
bool Exists(string strFileName)
```

FileInfo has four additional properties that reveal the attributes of the file and the dates the file was created, last accessed, and last written to:

FileInfo Properties (selection)

Type	Property	Accessibility
FileAttributes	*Attributes*	get/set
DateTime	*CreationTime*	get/set
DateTime	*LastAccessTime*	get/set
DateTime	*LastWriteTime*	get/set

These properties, all of which are inherited from *FileSystemInfo*, are all duplicated by static methods in the *File* class:

File Static Methods (selection)

```
FileAttributes GetAttributes(string strFileName)
DateTime GetCreationTime(string strFileName)
DateTime GetLastAccessTime(string strFileName)
DateTime GetLastWriteTime(string strFileName)
void SetAttributes(string strFileName, FileAttributes fa)
void SetCreationTime(string strFileName, DateTime dt)
void SetLastAccessTime(string strFileName, DateTime dt)
void SetLastWriteTime(string strFileName, DateTime dt)
```

The following methods let you copy, move, or delete the file. I've included the *Refresh* method here, which refreshes the object's properties after you've made a change to the file:

FileInfo Methods (selection)

```
FileInfo CopyTo(string strFileName)
FileInfo CopyTo(string strFileName, bool bOverwrite)
void MoveTo(string strFileName)
void Delete()
void Refresh()
```

The copy, move, and delete facilities are duplicated in the *File* class:

File Static Methods (selection)

```
void Copy(string strFileNameSrc, string strFileNameDst)
void Copy(string strFileNameSrc, string strFileNameDst, bool bOverwrite)
void Move(string strFileNameSrc, string strFileNameDst)
void Delete(string strFileName)
```

And finally, the *File* and *FileInfo* classes have several methods to open files:

FileInfo Methods (selection)

```
FileStream Create()
FileStream Open(FileMode fm)
FileStream Open(FileMode fm, FileAccess fa)
FileStream Open(FileMode fm, FileAccess fa, FileShare fs)
FileStream OpenRead()
FileStream OpenWrite()
StreamReader OpenText()
StreamWriter CreateText()
StreamWriter AppendText()
```

These are handy if you've just obtained an array of *FileInfo* objects from a *GetFiles* call on a *DirectoryInfo* object and you want to poke your nose into each and every file.

You can also use the corresponding static methods implemented in the *File* class:

File Static Methods (selection)

```
FileStream Create(string strFileName)
FileStream Open(string strFileName, FileMode fm)
FileStream Open(string strFileName, FileMode fm, FileAccess fa)
FileStream Open(string strFileName, FileMode fm, FileAccess fa, FileShare
fs)
FileStream OpenRead(string strFileName)
FileStream OpenWrite(string strFileName)
StreamReader OpenText(string strFileName)
StreamWriter CreateText(string strFileName)
StreamWriter AppendText(string strFileName)
```

However, these methods don't provide any real advantage over using the appropriate constructors of the *FileStream*, *StreamReader*, or *StreamWriter* class. Indeed, their very presence in the *File* class was initially one of the aspects of the entire *System.IO* namespace that I found most confusing. It doesn't make sense to use a class like *File* merely to obtain an object of type *FileStream* so that you can then use *FileStream* properties and methods. It's easier to use just a single class if that's sufficient for your purposes.