

Welcome to the C# Programming Primer.

Sams Publishing and Borland have teamed up to provide you with C# content from popular Sams books, as well as a special coupon offer for Borland customers.

These chapters cover the basics of programming with C# on the .NET platform, along with ASP.NET Web programming and more advanced C# techniques.

For a limited time only, you can purchase any of Sams C# and C#Builder titles **for 35% off the cover price.**

To redeem your coupon, go to:

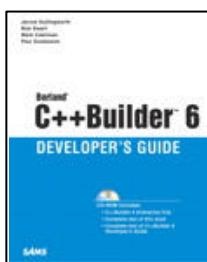
- 1) www.informit.com/sales/ad/Borlandoffer
- 2) Select from the book titles listed below
- 3) at checkout, type in the coupon code: BORLAND35 to receive your discount

The following titles have been selected for this coupon offer:

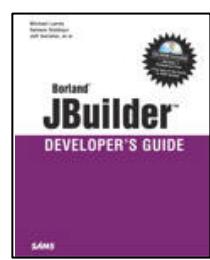


(New title coming this October! Pre-order with this coupon)

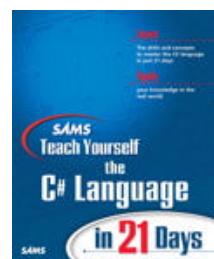
0-672-32589-6, Mayo, 10/8/2003, \$34.99



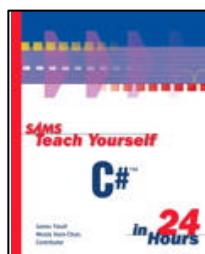
**Borland C++ Builder 6
Developer's Guide**
\$59.99



**Borland
JBuilder
Developer's
Guide**
\$59.99



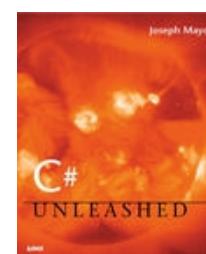
**Sams Teach Yourself the C#
Language in 21 Days**
\$34.99



**Sams Teach
Yourself C# in
24 Hours**
\$29.99



**Sams Teach Yourself C# in 21
Days**
\$39.99



C# Unleashed
\$49.95



C# Primer Plus
\$44.99

Contents

Sams Teach Yourself C# in 21 Days

DAY 1 Getting Started with C#	1
DAY 2 Understanding C# Programs	22
DAY 3 Storing Information with Variables	42
DAY 4 Working with Operators	72
DAY 5 Control Statements	102

C# PRIMER PLUS

CHAPTER 3 A GUIDED TOUR THROUGH C#: PART I	130
CHAPTER 4 A GUIDED TOUR THROUGH C#: PART II	161
CHAPTER 5 YOUR FIRST OBJECT-ORIENTED C# PROGRAM	181

C# and the .net framework

CHAPTER 3.2 User Interface Components	235
CHAPTER 3.3 Data Bound Controls	296
CHAPTER 3.4 Windows Forms Example Application (Scribble .NET)	316
CHAPTER 3.5 GDI+: The .NET Graphics Interface	350
CHAPTER 3.6 Practical Windows Forms Applications	394

Sams Teach Yourself C# Web Programming in 21 Days

Day 13 Introducing Web Services	441
Day 14 Publishing Web Services	461
Day 15 Consuming a Web Service	474
Day 16 Putting It All Together with Web Services	499

C# Unleashed

CHAPTER 23 Multi-Threading	520
CHAPTER 25 String Manipulation	526
CHAPTER 28 Reflection	556
CHAPTER 31 Runtime Debugging	570

WEEK 1

DAY 1

Getting Started with C#

Welcome to *Sams Teach Yourself C# in 21 Days!* In today's lesson you begin the process of becoming a proficient C# programmer. Today you

- Learn why C# is a great programming language to use
- Discover the steps in the Program Development Cycle
- Understand how to write, compile, and run your first C# program
- Explore error messages generated by the compiler and linker
- Review the types of applications that can be created with C#

What Is C#?

It would be unusual if you bought this book without knowing what C# is. It would not, however, be unusual if you didn't know a lot about the language. Released to the public in June 2000, C#—pronounced See Sharp—has not been around for very long.

C# is a new language created by Microsoft and submitted to the ECMA for standardization. This new language was created by a team of people at Microsoft led by Anders Hejlsberg. Interestingly, Hejlsberg is a Microsoft

Distinguished Engineer who has created other products and languages, including Borland Turbo C++ and Borland Delphi. With C#, he focused on taking what was right about existing languages and adding improvements to make something better.

C# is a powerful and flexible programming language. Like all programming languages, it can be used to create a variety of applications. Your potential with C# is limited only by your imagination. The language does not place constraints on what you can do. C# has already been used for projects as diverse as dynamic Web sites, development tools, and even compilers.

C# was created as an object-oriented programming (OOP) language. Other programming languages include object-oriented features, but very few are fully object-oriented. Later in today's lesson you will understand how C# compares to some of these other programming languages. You'll also learn what types of applications can be created. On Day 2, "Understanding C# Programming," you will learn what it means to use an object-oriented language.

Why C#?

Many people believed that there was no need for a new programming language. Java, C++, Perl, Microsoft Visual Basic, and other existing languages were believed to offer all the functionality needed.

C# is a language derived from C and C++, but it was created from the ground up. Microsoft started with what worked in C and C++ and included new features that would make these languages easier to use. Many of these features are very similar to what can be found in Java. Ultimately, Microsoft had a number of objectives when building the language. These objectives can be summarized in the claims Microsoft makes about C#:

- C# is simple.
- C# is modern.
- C# is object-oriented.

In addition to Microsoft's reasons, there are other reasons to use C#:

- C# is powerful and flexible.
- C# is a language of few words.
- C# is modular.
- C# will be popular.



The following section contains a lot of technical terms. Don't worry about understanding these. Most of them don't matter to C# programmers! The ones that do matter will be explained later in this book.

C# Is Simple

C# removes some of the complexities and pitfalls of languages such as Java and C++, including the removal of macros, templates, multiple inheritance, and virtual base classes. These are all areas that cause either confusion or potential problems for C++ developers. If you are learning C# as your first language, rest assured—these are topics you won't have to spend time learning!

C# is simple because it is based on C and C++. If you are familiar with C and C++—or even Java—you will find C# very familiar in many aspects. Statements, expressions, operators, and other functions are taken directly from C and C++, but improvements make the language simpler. Some of the improvements include eliminating redundancies. Other areas of improvement include additional syntax changes. For example, C++ has three operators for working with members: ::, ., and ->. Knowing when to use each of these three symbols can be very confusing in C++. In C#, these are all replaced with a single symbol—the “dot” operator. For newer programmers, this and many other features eliminate a lot of confusion.



If you have used Java and you believe it is simple, you will find C# to be simple. Most people don't believe that Java is simple. C# is, however, easier than Java and C++.

C# Is Modern

What makes a modern language? Features such as exception handling, garbage collection, extensible data types, and code security are features that are expected in a modern language. C# contains all of these. If you are a new programmer, you might be asking what all these complicated-sounding features are. By the end of your twenty-one days, you will understand how they all apply to your C# programming!



Pointers are an integral part of C and C++. They are also the most confusing part of the languages. C# removes much of the complexity and trouble caused by pointers. In C#, automatic garbage collection and type safety are

an integral part of the language. If you are not familiar with the concepts of pointers, garbage collection, and type safety, don't worry. These are all explained in later lessons.

C# Is Object-Oriented

NEW TERM The keys to an object-oriented language are encapsulation, inheritance, and polymorphism. C# supports all of these. *Encapsulation* is the placing of functionality into a single package. *Inheritance* is a structured way of extending existing code and functionality into new programs and packages. *Polymorphism* is the capability of adapting to what needs to be done. Detailed explanations of each of these terms and a more detailed description of object orientation are provided in Day 2's lesson. Additionally, these topics are covered in greater detail throughout this book.

C# Is Powerful and Flexible

As mentioned before, with C# you are limited only by your imagination. The language places no constraints on what can be done. C# can be used for projects as diverse as creating word processors, graphics, spreadsheets, and even compilers for other languages.

C# Is a Language of Few Words

NEW TERM C# is a language that uses a limited number of words. C# contains only a handful of terms, called *keywords*, which serve as the base on which the language's functionality is built. Table 1.1 lists the C# keywords. A majority of these keywords are used to describe information. You might think that a language with more keywords would be more powerful. This isn't true. As you program with C#, you will find that it can be used to do any task.

TABLE 1.1 The C# Keywords

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public

TABLE 1.1 continued

readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

1

Note

There are a few other words used in C# programs. While not keywords, they should be treated as though they were. Specifically, get, set, and value.

C# Is Modular

NEW TERM C# code can (and should) be written in chunks called *classes*, which contain routines called *member methods*. These classes and methods can be reused in other applications or programs. By passing pieces of information to the classes and methods, you can create useful, reusable code.

Note

On Day 2, you learn about classes, and on Day 6, “Classes,” you learn how to start creating your own.

C# Will Be Popular

C# is one of the newest programming languages. At the time this book was written, it was unknown as to what the popularity of C# would be, but it is a good bet that this will become a very popular language for a number of reasons. One of the key reasons is Microsoft and the promises of .NET.

Microsoft wants C# to be popular. Although a company cannot make a product be popular, it can help. Not long ago, Microsoft suffered the abysmal failure of the Microsoft Bob operating system. Although Microsoft wanted Bob to be popular, it failed.

C# stands a better chance of success than Microsoft Bob. I don’t know whether people at Microsoft actually used Bob in their daily jobs. C#, however, is being used by Microsoft. Many of its products have already had portions rewritten in C#. By using it, Microsoft helps validate the capabilities of C# to meet the needs of programmers.

Microsoft .NET is another reason why C# stands a chance to succeed. .NET is a change in the way the creation and implementation of applications is done. Although virtually

any programming language can be used with .NET, C# is proving to be the language of choice. Tomorrow's lesson includes a section that explains the high points of .NET.

C# will also be popular for all the features mentioned earlier: simplicity, object-orientation, modularity, flexibility, and conciseness.

C# Versus Other Programming Languages

You might have heard about Visual Basic, C++, and Java. Perhaps you're wondering what the differences are between C# and these other programming languages. You might also be wondering whether you should be teaching yourself one of these three languages instead of C#.



Note

The top questions on Internet discussion forums related to .NET are

- What is the difference between Java and C#?
- Isn't C# just a Java clone?
- What is the difference between C# and C++?
- Which should I learn, Visual Basic .NET or C#?

Microsoft says that C# brings the power of C++ with the ease of Visual Basic. C# does bring a lot of power, but is it as easy as Visual Basic? It might not be as easy as Visual Basic 6, but it is as easy as Visual Basic .NET (version 7), which was rewritten from the ground up. The end result is that Visual Basic is really no easier than programming C#. In fact, you can actually write many programs with less code using C#.

Although C# removes some of the features of C++ that cause programmers a lot of grief, no power or functionality was really lost. Some of the programming errors that are easy to create in C++ can be totally avoided in C#. This can save you hours or even days in finishing your programs. You'll understand more about the differences from C++ as you cover topics throughout this book.

Another language that has gotten lots of attention is Java. Java, like C++ and C#, is based on C. If you decide to learn Java later, you will find that a lot of what you learn about C# can be applied.

You might also have heard of the C programming language. Many people wonder if they should learn C before learning C#, C++, or Java. Simply put, there is absolutely no need to learn C first.

Enough about whys and wherefores. You most likely bought this book so you could learn to use the C# language to create your own programs. The following sections explore the

steps involved in creating a program. You then walk through the creation of a simple program from start to finish.

1

Preparing to Program

You should take certain steps when you're solving a problem. First, you must define the problem. If you don't know what the problem is, you can't find a solution! After you know what the problem is, you can devise a plan to fix it. When you have a plan, you can usually implement it. After the plan is implemented, you must test the results to see whether the problem is solved. This same logic can be applied to many other areas, including programming.

When creating a program in C# (or in any language), you should follow a similar sequence of steps:

1. Determine the objective(s) of the program.
2. Determine the methods you want to use in writing the program.
3. Create the program to solve the problem.
4. Run the program to see the results.

An example of an objective (see step 1) might be to write a word processor or database program. A much simpler objective is to display your name on the screen. If you don't have an objective, you won't be able to write an effective program.

The second step is to determine the method you want to use to write the program. Do you need a computer program to solve the problem? What information needs to be tracked? What formulas will be used? During this step, you should try to determine what will be needed and in what order the solution should be implemented.

As an example, assume that someone asks you to write a program to determine the area inside a circle. Step 1 is complete, because you know your objective: Determine the area inside a circle. Step 2 is to determine what you need to know to ascertain the area. In this example, assume that the user of the program will provide the radius of the circle.

Knowing this, you can apply the formula πr^2 to obtain the answer. Now you have the pieces you need, so you can continue to steps 3 and 4, which are called the Program Development Cycle.

The Program Development Cycle

The Program Development Cycle has its own steps. In the first step, you use an editor to create a file containing your source code. In the second step, you compile the source code to create an intermediate file called either an executable file or a library file. The third step is to run the program to see whether it works as originally planned.

Creating the Source Code

NEW TERM *Source code* is a series of statements or commands that are used to instruct the computer to perform your desired tasks. As mentioned, the first step in the Program Development Cycle is to enter source code into an editor. For example, here is a line of C# source code:

```
System.Console.WriteLine("Hello, Mom!");
```

This statement instructs the computer to display the message `Hello, Mom!` onscreen.
(For now, don't worry about how this statement works.)

Using an Editor

NEW TERM An *editor* is a program that can be used to enter and save source code. There are a number of editors that can be used with C#. Some are made specifically for C#, and others are not.

At the time this book was written, there were only a few editors created for C#; however, as time goes on, there will be many more. Microsoft has added C# capabilities to its Visual Studio product which includes Visual C#. This is the most predominant editor available. If you don't have Visual Studio .NET, however, you can still do C# programming.

There are also other editors available for C#. Like Visual Studio.NET, many of these enable you to do all the steps of the development cycle without leaving the editor. More importantly, most of these color-code the text you enter. This makes it much easier to find possible mistakes. Many editors will even help you by giving you information on what you need to enter and giving you a robust help system.

If you don't have a C# editor, don't fret. Most computer systems include a program that can be used as an editor. If you're using Microsoft Windows, you can use either Notepad or WordPad as your editor. If you're using a Linux or UNIX system, you can use such editors as ed, ex, edit, emacs, or vi.

Most word processors use special codes to format their documents. Other programs can't read these codes correctly. Many word processors—such as WordPerfect, Microsoft Word, and WordPad—are capable of saving source files in a text-based form. When you want to save a word processor's file as a text file, select the text option when saving.



To find alternative editors, you can check computer stores or computer mail-order catalogs. Another place to look is in the ads in computer programming magazines. The following are a few editors that were available at the time this book was written:

- **CodeWrite.** CodeWright is an editor that provides special support for ASP, XML, HTML, C#, Perl, Python, and more. It is located at www.premia.com.
- **EditPlus.** EditPlus is an Internet-ready text editor, HTML editor, and programmer's editor for Windows. Although it can serve as a good replacement for Notepad, it also offers many powerful features for Web page authors and programmers, including the color-coding of code. It is located at www.editplus.com.
- **JEdit.** JEdit is an Open-Source editor for Java; however, it can be used for C#. It includes the capability of color-coding the code. It is located at <http://jedit.sourceforge.net>.
- **Poorman IDE by Duncan Chen.** Poorman provides a syntax-highlighted editor for both C# and Visual Basic.NET. It also enables you to run the compiler and capture the console output so you don't need to leave the Poorman IDE. Poorman is located at www.geocities.com/duncanchen/poormanide.htm.
- **SharpDevelop by Mike Krüger.** SharpDevelop is a free editor for C# projects on Microsoft's .NET platform. It is an Open-Source Editor (GPL), so you can download both source code and executables from www.icsharpcode.net.

Naming Your Source Files

When you save a source file, you must give it a name that describes what the program does. In addition, when you save C# program source files, give the file a .cs extension. Although you could give your source file any name and extension, .cs is recognized as the appropriate extension to use.

Executing a C# Program

Before digging into the Program Development Cycle, it is important to understand a little bit about how a C# program executes. C# programs are different from programs you could create with other programming languages.

NEW TERM C# programs are created to run on the Common Language Runtime (CLR). This means that if you create a C# executable program and try to run it on a machine that doesn't have the CLR or a compatible runtime, it won't execute. *Executable* means that the program can be run, or executed, by your computer.

The benefit of creating programs for a runtime environment is portability. In older languages such as C and C++, if you wanted to create a program that could run on different platforms or operating systems, you had to compile different executable programs. For

example, if you wrote a C application and you wanted to run it on a Linux machine and a Windows machine, you would have to create two executable programs—one on a Linux machine and one on a Windows machine. With C#, you create only one executable program, and it runs on either machine.

NEW TERM If you want your program to execute as fast as possible, you want to create a true executable. A computer requires digital, or *binary*, instructions in what is called *machine language*. A program must be translated from source code to machine language. A program called a *compiler* performs this translation. The compiler takes your source code file as input and produces a disk file containing the machine language instructions that correspond to your source code statements. With programs such as C and C++, the compiler creates a file that can be executed with no further effort.

With C#, you use a compiler that does not produce machine language. Instead it produces an Intermediate Language (IL) file. Because this isn't directly executable by the computer, you need something more to happen to translate or further compile the program for the computer. The CLR or a compatible C# runtime does this final compile just as it is needed.

One of the first things the CLR does with an IL file is a final compile of the program. In this process, the CLR converts the code from the portable, IL code to a language (machine language) that the computer can understand and run. The CLR actually compiles only the parts of the program that are being used. This saves time. Additionally, after a portion of your IL file has been given a true compile on a machine, it never needs to be compiled again, because the final compiled portion of the program is saved and used the next time that portion of the program is executed.



Note

Because the runtime needs to compile the IL file, it takes a little more time to run a program the first time than it does to run a fully compiled language such as C++. After the first time a program is completely executed, the time difference disappears because the fully compiled version will be used from that point.



Note

The last minute compiling of a C# program is called Just In Time compiling or *jitting*.

Compiling C# Source Code

To create the IL file, you use the C# compiler. You typically use the `csc` command to run the compiler, followed by the name of the source file. For example, to compile a source file called `radius.cs`, you type the following at the command line:

```
csc radius.cs
```

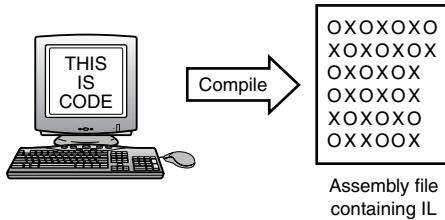
If you're using a graphical development environment, compiling is even simpler. In most graphical environments, you can compile a program by selecting the compile icon or selecting the appropriate option from the menu. After the code is compiled, selecting the run icon or selecting the appropriate option from the menus executes the program. You should check your compiler's manuals for specifics on compiling and running a program.

NEW TERM After you compile, you have an IL file. If you look at a list of the files in the directory or folder in which you compiled, you should find a new file that has the same name as your source file, but with an .exe (rather than a .cs) extension. The file with the .exe extension is your “compiled” program (called an *assembly*). This program is ready to run on the CLR. The assembly file contains all the information that the common runtime needs to know to execute the program.

Figure 1.1 shows the progression from source code to executable.

FIGURE 1.1.

The C# source code that you write is converted to Intermediate Language (IL) code by the compiler.



Note

In general, two types of deliverables are created as C# programs—executables and libraries. For the two weeks of this book you focus on executables, which are EXE files. You can also use C# for other types of programming, including scripting on ASP.NET pages. You learn about libraries in the third week.

Completing the Development Cycle

After your program is a compiled IL file, you can run it by entering its name at the command-line prompt or just as you would run any other program.

If you run the program and receive results different from what you thought you would, you need to go back to the first step of the development process. You must identify what caused the problem and correct it in the source code. When you make a change to the source code, you need to recompile the program to create a corrected version of the executable file. You keep following this cycle until you get the program to execute exactly as you intended.

The C# Development Cycle

- Step 1 Use an editor to write your source code. C# source code files are usually given the .cs extension (for example, a_program.cs, database.cs, and so on).
- Step 2 Compile the program using a C# compiler. If the compiler doesn't find any errors in the program, it produces an assembly file with the extension .exe or .dll. For example, myprog.cs compiles to myprog.exe by default. If the compiler finds errors, it reports them. You must return to step 1 to make corrections in your source code.
- Step 3 Execute the program on a machine with a C# runtime, such as the Common Language Runtime. You should test to determine whether your program functions properly. If not, start again with step 1 and make modifications and additions to your source code.

Figure 1.2 shows the program development steps. For all but the simplest programs, you might go through this sequence many times before finishing your program. Even the most experienced programmers can't sit down and write a complete, error-free program in just one step! Because you'll be running through the edit-compile-test cycle many times, it's important to become familiar with your tools: the editor, compiler, and runtime environment.

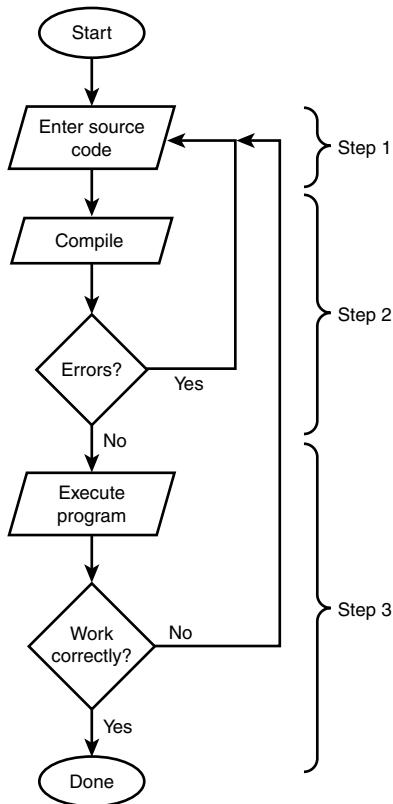
Your First C# Program

You're probably eager to try your first program in C#. To help you become familiar with your compiler, Listing 1.1 contains a quick program for you to work through. You might not understand everything at this point, but you should get a feel for the process of writing, compiling, and running a real C# program.

This demonstration uses a program named hello.cs, which does nothing more than display the words `Hello, World!` on the screen. This program is the traditional program used to introduce people to programming. It is also a good one for you to use to learn. The source code for hello.cs is in Listing 1.1. When you type this listing, don't include the line numbers on the left or the colons.

FIGURE 1.2.

The steps involved in C# program development.

**LISTING 1.1** hello.cs

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine("Hello, World!");
6:     }
7: }
```

Be sure that you have installed your compiler as specified in the installation instructions provided with the software. When your compiler and editor are ready, follow the steps in the next section to enter, compile, and execute hello.cs.

Entering and Compiling hello.cs

To enter and compile the hello.cs program, follow these steps:

1. Start your editor.
2. Use the keyboard to type the hello.cs source code shown in Listing 1.1. Don't enter the line numbers or colons. These are provided only for reference within this book. Press Enter at the end of each line. Make sure that you enter the code using the same case. C# is case sensitive, so if you change the capitalization, you will get errors.



If you are a C or C++ programmer, you will most likely make a common mistake. In C and C++, `main()` is lowercase. In C#, `Main()` has a capital M. In C#, if you type a lowercase `m`, you will get an error.

3. Save the source code. You should name the file `hello.cs`.
4. Verify that `hello.cs` has been saved by listing the files in the directory or folder.
5. Compile `hello.cs`. If you are using the command-line compiler, enter the following:

```
csc hello.cs
```

If you are using an Integrated Development Environment, select the appropriate icon or menu option. You should get a message stating that there were no errors or warnings.

6. Check the compiler messages. If you receive no errors or warnings, everything should be okay.

If you made an error typing the program, the compiler will catch it and display an error message. For example, if you misspelled the word `Console` as `Consol`, you would see a message similar to the following:

```
hello.cs(5,7): error CS0117: 'System' does not contain a definition for  
'Consol'
```

7. Go back to step 2 if this or any other error message is displayed. Open the `hello.cs` file in your editor. Compare your file's contents carefully with Listing 1.1, make any necessary corrections, and continue with step 3.
8. Your first C# program should now be compiled and ready to run. If you display a directory listing of all files named `hello` (with any extension), you should see the following:
`hello.cs`, the source code file you created with your editor
`hello.exe`, the executable program created when you compiled `hello.cs`

9. To *execute*, or run, hello.exe, enter hello at the command line. The message Hello, World! is displayed onscreen.

 **Note**

If you run the hello program by double-clicking in Microsoft's Windows Explorer, you might not see the results. This program runs in a command-line window. When you double-click in Windows Explorer, the program opens a command-line window, runs the program, and—because the program is done—closes the window. This can happen so fast that it doesn't seem that anything happens. It is better to open a command-line window, change to the directory containing the program, and then run the program from the command line.

Congratulations! You have just entered, compiled, and run your first C# program. Admittedly, hello.cs is a simple program that doesn't do anything useful, but it's a start. In fact, most of today's expert programmers started learning in this same way—by compiling a "hello world" program.

Understanding Compilation Errors

A compilation error occurs when the compiler finds something in the source code that it can't compile. A misspelling, typographical error, or any of a dozen other things can cause the compiler to choke. Fortunately, modern compilers don't just choke; they tell you what they're choking on and where the problem is. This makes it easier to find and correct errors in your source code.

This point can be illustrated by introducing a deliberate error into the hello.cs program you entered earlier. If you worked through that example (and you should have), you now have a copy of hello.cs on your disk. Using your editor, move the cursor to the end of line 5 and erase the terminating semicolon. hello.cs should now look like Listing 1.2.

LISTING 1.2 hello.cs with an Error

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine("Hello, World!")
6:     }
7: }
```

Next, save the file. You're now ready to compile it. Do so by entering the command for your compiler. Remember, the command-line command is

```
csc hello.cs
```

Because of the error you introduced, the compilation is not completed. Rather, the compiler displays a message similar to the following:

```
hello.cs(5,48): error CS1002: ; expected
```

Looking at this line, you can see that it has three parts:

hello.cs	The name of the file where the error was found
(5,48) :	The line number and position where the error was noticed: line 5, position 8
error CS1002: ; expected	A description of the error

This message is quite informative, telling you that when the compiler made it to the 48th character of line 5 of hello.cs, the compiler expected to find a semicolon but didn't. Although the compiler is very clever about detecting and localizing errors, it's no Einstein. Using your knowledge of the C# language, you must interpret the compiler's messages and determine the actual location of any errors that are reported. They are often found on the line reported by the compiler, but if not, they are almost always on the preceding line. You might have a bit of trouble finding errors at first, but you should soon get better at it.

Before leaving this topic, take a look at another example of a compilation error. Load hello.cs into your editor again and make the following changes:

1. Replace the semicolon at the end of line 5.
2. Delete the double quotation mark just before the word Hello.

Save the file to disk and compile the program again. This time, the compiler should display an error message similar to the following:

```
hello.cs(5,32): error CS1010: Newline in constant
```

The error message finds the location of the error correctly, locating it in line 5. The error messages found the error at location 32 on line 5. This location is the location of the first quote for the text to be displayed. This error message missed the point that there was a quotation mark missing from the code. In this case, the compiler took its best guess at the problem. Although it was close to the area of the problem, it was not perfect.

 Tip

If the compiler reports multiple errors, and you can find only one, fix that error and recompile. You might find that your single correction is all that's needed, and the program will compile without errors.

1

Understanding Logic Errors

There is one other type of error you might get: logic errors. Logic errors are not errors you can blame on the code or the compiler; they are errors that can be blamed only on you. It is possible to create a program with perfect C# code that still contains an error. For example, suppose you want to calculate the area of a circle by multiplying 2 multiplied by the value of PI multiplied by the radius:

$$\text{Area} = 2\pi r$$

You can enter this formula into your program, compile, and execute. You will get an answer. The C# program could be written syntactically correct; however, every time you run this program, you will get a wrong answer. The logic is wrong. This formula will never give you the area of a circle; it gives you its circumference. You should have used the formula πr^2 !

No matter how good a compiler is, it will never be able to find logic errors. You have to find these on your own by reviewing your code and by running your programs.

Types of C# Programs

Before ending today's lessons, it is worth knowing what types of applications you can create with C#. There are a number of types you can build:

- **Console applications.** Console applications run from the command line. Throughout this book you will create console applications, which are primarily character- or text-based and therefore remain relatively simple to understand.
- **Windows applications.** You can also create Windows applications that take advantage of the graphical user interface (GUI) provided by Microsoft Windows.
- **Web Services.** Web services are routines that can be called across the Web.
- **Web Form / ASP.NET applications.** ASP.NET applications are executed on a Web server and generate dynamic Web pages.

In addition to these types of applications, C# can be used to do a lot of other things, including creating libraries, creating controls, and more.

Summary

At the beginning of today's lesson you learned what C# has to offer, including its power, its flexibility, and its object orientation. You also learned that C# is considered simple and modern.

Today you explored the various steps involved in writing a C# program—the process known as program development. You should have a clear grasp of the edit-compile-test cycle before continuing.

Errors are an unavoidable part of program development. Your C# compiler detects errors in your source code and displays an error message, giving both the nature and the location of the error. Using this information, you can edit your source code to correct the error. Remember, however, that the compiler can't always accurately report the nature and location of an error. Sometimes you need to use your knowledge of C# to track down exactly what is causing a given error message.

Q&A

Q Will a C# program run on any machine?

A No. A C# program will run only on machines that have the Common Language Runtime (CLR) installed. If you copy the executable program to a machine that does not contain the CLR, you get an error. On versions of Windows without the CLR, you usually are told that a DLL file is missing.

Q If I want to give people a program I wrote, which files do I need to give them?

A One of the nice things about C# is that it is a compiled language. This means that after the source code is compiled, you have an executable program. If you want to give the hello program to all your friends with computers, you can. You give them the executable program, hello.exe. They don't need the source file, hello.cs, and they don't need to own a C# compiler. They do need to use a computer system that has a C# runtime, such as the Common Language Runtime from Microsoft.

Q After I create an executable file, do I need to keep the source file (.cs)?

A If you get rid of the source file, you have no easy way to make changes to the program in the future, so you should keep this file.

Most integrated development environments create files in addition to the source file (.cs) and the executable file. As long as you keep the source file (.cs), you can almost always re-create the other files. If your program uses external resources, such as images and forms, you also need to keep those files in case you need to make changes and re-create the executable.

Q If my compiler came with an editor, do I have to use it?

A Definitely not. You can use any editor, as long as it saves the source code in text format. If the compiler came with an editor, you should try to use it. If you like a different editor better, use it. I use an editor that I purchased separately, even though all my compilers have their own editors. The editors that come with compilers are getting better. Some of them automatically format your C# code. Others color-code different parts of your source file to make it easier to find errors.

Q Can I ignore warning messages?

A Some warning messages don't affect how the program runs, and some do. If the compiler gives you a warning message, it's a signal that something isn't right. Most compilers let you set the warning level. By setting the warning level, you can get only the most serious warnings, or you can get all the warnings, including the most minute. Some compilers even offer various levels between. In your programs, you should look at each warning and make a determination. It's always best to try to write all your programs with absolutely no warnings or errors. (With an error, your compiler won't create the executable file.)

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. Give three reasons why C# is a great choice of programming language.
2. What do IL and CLR stand for?
3. What are the steps in the Program Development Cycle?
4. What command do you need to enter to compile a program called my_prog.cs with your compiler?
5. What extension should you use for your C# source files?
6. Is filename.txt a valid name for a C# source file?
7. If you execute a program that you have compiled and it doesn't work as you expected, what should you do?
8. What is machine language?

9. On what line did the following error most likely occur?

```
my_prog.cs(35,6): error CS1010: Newline in constant
```

10. Near what column did the following error most likely occur?

```
my_prog.cs(35,6): error CS1010: Newline in constant
```

Exercises

1. Use your text editor to look at the EXE file created by Listing 1.1. Does the EXE file look like the source file? (Don't save this file when you exit the editor.)
2. Enter the following program and compile it. (Don't include the line numbers or colons.) What does this program do?

```
1: // circle.cs - Using variables and literals
2: // This program calculates some circle stuff.
3: //-----
4:
5: using System;
6:
7: class variables
8: {
9:     public static void Main()
10:    {
11:        //Declare variables
12:
13:        int radius = 4;
14:        const double PI = 3.14159;
15:        double circum, area;
16:
17:        //Do calculations
18:
19:        area = PI * radius * radius;
20:        circum = 2 * PI * radius;
21:
22:        //Print the results
23:
24:        Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:        Console.WriteLine("The area is {0}", area);
26:        Console.WriteLine("The circumference is {0}", circum);
27:    }
28: }
```

3. Enter and compile the following program. What does this program do?

```
1: class AClass
2: {
3:     static void Main()
4:     {
5:         int x,y;
6:         for ( x = 0; x < 10; x++, System.Console.Write( "\n" ) )
```

```
7:         for ( y = 0; y < 10; y++ )
8:             System.Console.Write( "X" );
9:         }
10:    }
```

4. **BUG BUSTER:** The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine(Keep Looking!);
6:         System.Console.WriteLine(You'll find it!);
7:     }
8: }
```

5. Make the following change to the program in exercise 3. Recompile and rerun this program. What does the program do now?

```
8:         System.Console.Write( "{0}" , (char) 1 );
```

WEEK 1

DAY 2

Understanding C# Programs

In addition to understanding the basic composition of a program, you also need to understand the structure of creating a C# program. Today you

- Learn about the parts of a C# application
- Understand C# statements and expressions
- Discover the facts about object-oriented programming
- Examine encapsulation, polymorphism, inheritance, and reuse
- Display basic information in your programs

C# Applications

The first part of today's lesson focuses on a simple C# application. Using Listing 2.1, you will gain an understanding of some of the key parts of a C# application.

LISTING 2.1 app.cs—Example C# Application

```
1: // app.cs - A sample C# application
2: // Don't worry about understanding everything in
3: // this listing. You'll learn all about it later!
4: //-----
5:
6: using System;
7:
8: class sample
9: {
10:     public static void Main()
11:     {
12:         //Declare variables
13:
14:         int radius = 4;
15:         const double PI = 3.14159;
16:         double area;
17:
18:         //Do calculation
19:
20:         area = PI * radius * radius;
21:
22:         //Print the results
23:
24:         Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:         Console.WriteLine("The area is {0}", area);
26:     }
27: }
```

You should enter this listing into your editor and then use your compiler to create the program. You can save the program as `app.cs`. When compiling the program, you enter the following at the command prompt:

`csc app.cs`

Alternatively, if you are using a visual editor, you should be able to select a compiler from the menu options.



Remember, you don't enter the line numbers or the colons when you are entering the listing above. The line numbers are to help discuss the listing in the lessons.

When you run the program, you get the following output:

OUTPUT Radius = 4, PI = 3.14159
The area is 50.3344

As you can see, the output from this listing is pretty straightforward. The value of a radius and the value of PI are displayed. The area of a circle based on these two values is then displayed.

In the following sections you are going to learn about some of the different parts of this program. Don't worry about understanding everything. In the lessons presented on later days, you will be revisiting this information in much greater detail. The purpose of the following sections is to give you a first look.

2

Comments

The first four lines of Listing 2.1 are comments. Comments are used to enter information in your program that can be ignored by the compiler. Why would you want to enter information that the compiler will ignore? There are a number of reasons.

Comments are often used to provide descriptive information about your listing—for example, identification information. Additionally, by entering comments, you can document what a listing is expected to do. Even though you might be the only one to use a listing, it is still a good idea to put in information about what the program does and how it does it. Although you know what the listing does now—because you just wrote it—you might not be able to remember later what you were thinking. If you give your listing to others, the comments will help them understand what the code was intended to do. Comments can also be used to provide revision history of a listing.

The main thing to understand about comments is that they are for programmers using the listing. The compiler actually ignores them. In C#, there are three types of comments you can use:

- One-line comments
- Multiline comments
- Documentation comments

Tip

Comments are removed by the compiler. Because they are removed, there is no penalty for having them in your program listings. If in doubt, you should include a comment.

One-Line Comments

Listing 2.1 uses one-line comments in each of lines 1 through 4. Lines 12, 18, and 22 also contain one-line comments. One-line comments have the following format:

```
// comment text
```

The two slashes indicate that a comment is beginning. From that point to the end of the current line, everything is treated as a comment.

A one-line comment does not have to start at the beginning of the line. You can actually have C# code on the line before the comments; however, after the two forward slashes, the rest of the line is a comment.

Multiline Comments

Listing 2.1 does not contain any multiline comments, but there are times when you want a comment to go across multiple lines. In this case you can either start each line with the double forward slash (as in lines 1 to 4 of the listing), or you can use multiline comments.

Multiline comments are created with a starting and ending token. To start a multiline comment, you enter a forward slash followed by an asterisk:

```
/*
```

Everything after that token is a comment until you enter the ending token. The ending token is an asterisk followed by a forward slash:

```
*/
```

The following is a comment:

```
/* this is a comment */
```

The following is also a comment:

```
/* this is  
a comment that  
is on  
a number of  
lines */
```

You can also enter this comment as the following:

```
// this is  
// a comment that  
// is on  
// a number of  
// lines
```

The advantage of using multiline comments is that you can “comment out” a section of a code listing by simply adding the /* and */. Anything that appears between the /* and the */ is ignored by the compiler as a comment.



Caution

You cannot nest multiline comments. This means that you cannot place one multiline comment inside of another. For example, the following is an error:

```
/* Beginning of a comment...
 * with another comment nested */
*/
```

2

Documentation Comments

C# has a special type of comment that enables you to create external documentation automatically.

These comments are identified with three slashes instead of the two used for single-line comments. These comments also use Extensible Markup Language (XML) style tags. XML is a standard used to mark up data. Although any valid XML tag can be used, common tags used for C# include <c>, <code>, <example>, <exception>, <list>, <para>, <param>, <paramref>, <permission>, <remarks>, <returns>, <see>, <seealso>, <summary>, and <value>.

These comments are placed in your code listings. Listing 2.2 shows an example of these comments being used. You can compile this listing as you have earlier listings. See Day 1, “Getting Started with C#,” if you need a refresher.

LISTING 2.2 xmlapp.cs—Using XML Comments

```
1: // xmlapp.cs - A sample C# application using XML
2: //           documentation
3: //-----
4:
5: /// <summary>
6: /// This is a summary describing the class.</summary>
7: /// <remarks>
8: /// This is a longer comment that can be used to describe
9: /// the class. </remarks>
10: class myApp
11: {
12:     /// <summary>
13:     /// The entry point for the application.
14:     /// </summary>
15:     /// <param name="args"> A list of command line arguments</param>
16:     public static void Main(string[] args)
```

LISTING 2.2 continued

```
17:      {  
18:          System.Console.WriteLine("An XML Documented Program");  
19:      }  
20:  }
```

When you compile and execute this listing, you get the following output:

OUTPUT

An XML Documented Program

To get the XML documentation, you must compile this listing differently from what you have seen before. To get the XML documentation, add the /doc parameter when you compile at the command line. If you are compiling at the command line, you enter

```
csc /doc:xmlfile xmlapp.cs
```

When you compile, you get the same output as before when you run the program. The difference is that you also get a file called `xmlfile` that contains documentation in XML. You can replace `xmlfile` with any name you'd like to give your XML file. For Listing 2.2, the XML file is

```
<?xml version="1.0"?>  
<doc>  
    <assembly>  
        <name>xmlapp</name>  
    </assembly>  
    <members>  
        <member name="T:myApp">  
            <summary>  
                This is a summary describing the class.</summary>  
            <remarks>  
                This is a longer comment that can be used to describe  
                the class. </remarks>  
            </member>  
        <member name="M:myApp.Main(System.String[])">  
            <summary>  
                The entry point for the application.  
            </summary>  
            <param name="args"> A list of command line arguments</param>  
        </member>  
    </members>  
</doc>
```

Note

It is beyond the scope of this book to cover XML and XML files.

 Note

If you are using a tool such as Visual Studio.NET, you need to check the documentation or help system to learn how to generate the XML documentation. Even if you are using such a tool, you should still be able to compile your programs from the command line.

2

Basic Parts of a C# Application

A programming language is composed of a bunch of words combined. A computer program is the formatting and use of these words in an organized manner. The parts of a C# language include the following:

- Whitespace
- C# keywords
- Literals
- Identifiers

Whitespace

NEW TERM If you look at Listing 2.1, you can see that it has been formatted so that the code lines up and is relatively easy to read. The blank spaces put into a listing are called *whitespace*. The basis of this term is that on white paper, you wouldn't see the spaces. Whitespace can consist of spaces, tabs, line feeds, and carriage returns.

The compiler almost always ignores whitespace. Because of this, you can add as many spaces, tabs, or line feeds as you want. For example, consider line 14 from Listing 2.1:

```
int radius = 4;
```

This is a well-formatted line with a single space between items. This line could have had additional spaces:

```
int      radius      =      4      ;
```

This line with extra spaces executes the same way as the original. In fact, when the program is run through the C# compiler, the extra whitespace is removed. You could also format this code across multiple lines:

```
int  
radius  
=  
4  
;
```

Although this is not very readable, it still works.

Because whitespace is ignored in general usage, you should make liberal use of it to help format your code and make it readable.

The exception to the compiler ignoring whitespace has to do with the use of text within quotation marks. When you use text within double quotes, whitespace is important because the text is to be used exactly as presented. Text has been used within quotation marks with the listings you have seen so far. In Listing 2.1, lines 24 and 25 contain text within double quotes. This text is written exactly as it is presented between the quotation marks.



Tip

Use whitespace to make your code easier to read.

C# Keywords

Recall from Day 1 that keywords are specific terms that have special meaning and therefore make up a language. The C# language has a number of keywords, which are listed in Table 1.1.

These keywords have a specific meaning when you program in C#. You will learn the meaning of these as you work through this book. Because all these words have a special meaning, they are reserved. You cannot use them for your own use. If you compare the words in Table 1.1 to Listing 2.1 or any of the other listings in this book, you will see that much of the listing is composed of keywords.



Note

Appendix B, "C# Keywords," contains short definitions for each of the C# keywords.

Literals

NEW TERM

Literals are straightforward hard-coded values. They are literally what they are!

For example, the numbers 4 and 3.14159 are both literals. Additionally, the text within double quotes is literal text. In tomorrow's lesson, you will learn more details on literals and their use.

Identifiers

NEW TERM In addition to C# keywords and literals, you have other words that are used within C# programs. These words are considered *identifiers*. In Listing 2.1 there are a number of identifiers, including `System` in line 6; `sample` in line 8; `radius` in line 14; `PI` in line 15; `area` in line 16; and `PI`, `radius`, and `area` in line 22.

Structure of a C# Application

2

Words and phrases are used to make sentences and sentences are used to make paragraphs. In the same way, whitespace, keywords, literals, and identifiers are combined to make expressions and statements. These in turn are combined to make a program.

C# Expressions and Statements

NEW TERM *Expressions* are like phrases. They are snippets of code made up of keywords. For example, the following are simple expressions:

```
PI = 3.14159  
PI * radius * radius
```

Statements are like sentences. They complete a single thought. A statement generally ends with a punctuation character—a semicolon (;). In Listing 2.1, lines 14, 15, and 16 are examples of statements.

The Empty Statement

One general statement deserves special mention: the empty statement. As you learned previously, statements generally end with a semicolon. You can actually put a semicolon on a line by itself. This is a statement that does nothing. Because there are no expressions to execute, the statement is considered an empty statement.

You might be wondering why you would want to include a statement that does nothing. Although this might not seem like something you would do, by the time you finish this book, you will find that an empty statement is valuable. You explore one of the most prevalent uses of an empty statement on Day 5, “Control Statements.”

Analysis of Listing 2.1

ANALYSIS It is worth taking a closer look at Listing 2.1 now that you’ve learned of some of the many concepts. The following sections review each line of Listing 2.1.

Lines 1–4—Comments

As you already learned, lines 1–4 contain comments that will be ignored by the compiler. These are for you and anyone who reviews the source code.

Lines 5, 7, 13, 17, 21, and 23—Whitespace

Line 5 is blank. You learned that a blank line is simply whitespace that will be ignored by the compiler. This line is included to make the listing easier to read. Lines 7, 13, 17, 21, and 23 are also blank. You can remove these lines from your source file and there will be no difference in how your program runs.

Line 6—The `using` Statement

Line 6 is a statement that contains the keyword `using` and a literal `System`. As with most statements, this ends with a semicolon. The `using` keyword is used to condense the amount of typing you need to do in your listing. Generally, the `using` keyword is used with namespaces. Namespaces and details on the `using` keyword are covered in some detail on Day 6, “Classes.”

Line 8—Class Declaration

C# is an object-oriented programming {OOP} language. Object-oriented languages use classes to declare objects. This program defines a class called `sample`. You will understand classes by the time you complete this book, and will get an overview of classes later in today’s lesson. The C# details concerning classes start on Day 6.

Lines 9, 11, 26, and 27—Punctuation Characters

Line 9 contains an opening braces that is paired with a closing brace in line 27. Line 11 has an opening brace that is paired with the closing one in line 26. These sets of braces contain and organize blocks of code. As you learn about different commands over the next four days, you will see how these braces are used.

Line 10—`Main()`

NEW TERM The computer needs to know where to start executing a program. C# programs start executing with the `Main()` function, as in line 10. A *function* is a grouping of code that can be executed by calling the function’s name. You’ll learn the details about functions on Day 7, “Class Methods and Member Functions.” The `Main()` function is special because it is used as a starting point.



The case of your code—whether letters are capitalized—is critical in C# applications. `Main()` has only the *M* capitalized. C++ and C programmers need to be aware that `main()`—in lowercase—does not work for C#.

2

Lines 14, 15, and 16—Declarations

Lines 14, 15, and 16 contain statements used to create identifiers that will store information. These identifiers are used later to do calculations. Line 14 declares an identifier to store the value of a radius. The literal 4 is assigned to this identifier. Line 15 creates an identifier to store the value of PI. This identifier, `PI`, is filled with the literal value of 3.14159. Line 16 declares an identifier that is not given any value.

Line 20—The Assignment Statement

Line 20 contains a simple statement that multiplies the identifier `PI` by the radius twice. The result of this expression is then assigned to the identifier `area`.

Lines 24 and 25—Calling Functions

Lines 24 and 25 are the most complex expressions in this listing. These two lines call a predefined routine that prints information to the console (screen). You will learn about these predefined functions later in today’s lesson.

Object-Oriented Programming (OOP)

As mentioned earlier, C# is considered an object-oriented language. To take full advantage of C#, you should understand the concepts of object-oriented languages. The following sections present an overview about objects and what makes a language object-oriented. You will learn how these concepts are applied to C# as you work through the rest of this book.

Object-Oriented Concepts

What makes a language object-oriented? The most obvious answer is that the language uses objects! This, however, doesn’t tell you much. Recall from Day 1 that there are three concepts generally associated with object-oriented languages:

- Encapsulation
- Polymorphism
- Inheritance

There is a fourth concept that is expected as a result of using an object-oriented language: reuse.

Encapsulation

Encapsulation is the concept of making “packages” that contain everything you need. With object-oriented programming, this means that you could create an object (or package) such as a circle that would do everything that you would want to do with a circle. This includes keeping track of everything about the circle, such as the radius and center point. It also means knowing how to do the functionality of a circle, such as calculating its radius and possibly knowing how to draw it.

By encapsulating a circle, you allow the user to be oblivious to how the circle works. You only need to know how to interact with the circle. This provides a shield to the inner workings of the circle. Why should users care how information about a circle is stored internally? As long as they can get the circle to do what they want, they shouldn’t.

Polymorphism

Polymorphism is the capability of assuming many forms. This can be applied to two areas of object-oriented programming (if not more). First, it means you can call an object or a routine in many different ways and still get the same result. Using a circle as an example, you might want to call a circle object to get its area. You can do this by using three points or by using a single point and the radius. Either way, you would expect to get the same results. In a procedure language such as C, you need two routines with two different names to address these two methods of getting the area. In C#, you still have two routines; however, you can give them the same name. Any programs you or others write will simply call the circle routine and pass your information. The circle program automatically determines which of the two routines to use. Based on the information passed, the correct routine will be used. Users calling the routine don’t need to worry about which routine to use. They just call the routine.

Polymorphism is also the ability to work with multiple forms. You will learn more about this form of polymorphism on Day 11, “Inheritance.”

Inheritance

Inheritance is the most complicated of the object-oriented concepts. Having a circle is nice, but what if a sphere would be nicer? A sphere is just a special kind of circle. It has all the characteristics of a circle with a third dimension added. You could say that a sphere is a special kind of circle that takes on all the properties of a circle and then adds a little more. By using the circle to create your sphere, your sphere can inherit all the properties of the circle. The capability of inheriting these properties is a characteristic of inheritance.

Reuse

One of the key reasons an object-oriented language is used is the concept of reuse. When you create a class, you can reuse it to create lots of objects. By using inheritance and some of the features described previously, you can create routines that can be used again in a number of programs and in a number of ways. By encapsulating functionality, you can create routines that have been tested and proven to work. This means you won't have to test the details of how the functionality works, only that you are using it correctly. This makes reusing these routines quick and easy.

Objects and Classes

NEW TERM

Now that you understand the concepts of an object-oriented language, it is important to understand the difference between a class and an object. A *class* is a definition for an item that will be created. The actual item that will be created is an *object*. Simply put, classes are definitions used to create objects.

An analogy often used to describe classes is a cookie cutter. A cookie cutter defines a cookie shape. It isn't a cookie, and it isn't edible. It is simply a construct that can be used to create shaped cookies over and over. When you use the cookie cutter to create cookies, you know that each cookie is going to look the same. You also know that you can use the cookie cutter to create lots and lots of cookies.

As with a cookie cutter, a class can be used to create lots of objects. For example, you can have a circle class that can be used to create a number of circles. If you create a drawing program to draw circles, you could have one circle class and lots of circle objects. You could make each circle in the snowman an object; however, you would need only one class to define all of them.

You also can have a number of other classes, including a name class, a card class, an application class, a point class, a circle class, an address class, a snowman class (that can use the circle class), and more.

**Note**

Classes and objects are covered again in more detail starting on Day 6. Today's information gives you an overview of the object-oriented concepts.

Displaying Basic Information

To help you have a little more fun early in this book, let's look at two routines that you can use to display information. When you understand these two routines, you will be able to display some basic information.

The two routines that you will be seeing used throughout this book to display basic information are

- `System.Console.WriteLine()`
- `System.Console.Write()`

These routines print information to the screen. Both print information in the same manner, with only one small difference. The `WriteLine()` routine writes information to a new line. The `Write()` routine does not start a new line when information is written.

The information you are going to display on the screen is written between the parentheses. If you are printing text, you include the text between the parentheses and within double quotes. For example, the following prints the text “Hello World”:

```
System.Console.WriteLine("Hello World");
```



Note

You used this routine on Day 1.

This prints `Hello World` on the screen. The following examples illustrate other text being printed:

```
System.Console.WriteLine("This is a line of text");
System.Console.WriteLine("This is a second line of text");
```

If you execute these consecutively, you see the following displayed:

```
This is a line of text
This is a second line of text
```

Now consider the following two lines. If these execute consecutively, what do you see printed?

```
System.Console.WriteLine("Hello ");
System.Console.WriteLine("World!");
```

If you guessed that these would print

```
Hello World!
```

you are not correct! Those lines print the following:

```
Hello
World!
```

Notice that each word is on a separate line. If you execute the two lines using the `Write()` routine instead, you get the results you want:

Hello World!

As you can see, the difference between the two routines is that `WriteLine()` automatically goes to a new line after the text is displayed, whereas `Write()` does not. Listing 2.3 shows the two routines in action.

LISTING 2.3 display.cs—Using `WriteLine()` and `Write()`.

```
1: // display.cs - printing with WriteLine and Write
2: //-----
3:
4: class display
5: {
6:     public static void Main()
7:     {
8:         System.Console.WriteLine("First WriteLine Line");
9:         System.Console.WriteLine("Second WriteLine Line");
10:
11:        System.Console.Write("First Write Line");
12:        System.Console.Write("Second Write Line");
13:
14:        // Passing parameters
15:        System.Console.WriteLine("\nWriteLine: Parameter = {0}", 123 );
16:
17:        System.Console.Write("Write: Parameter = {0}", 456);
18:    }
19: }
```

Remember that to compile this listing from the command line, you enter the following:

`csc display.cs`

If you are using an integrated development tool, you can select the compile option.

OUTPUT

```
First WriteLine Line
Second WriteLine Line
First Write LineSecond Write Line
WriteLine: Parameter = 123
Write: Parameter = 456
```

ANALYSIS

This listing uses the `System.Console.WriteLine()` routine on lines 8 and 9 to print two pieces of text. You can see from the output that each of these print on a separate line. Lines 11 and 12 show the `System.Console.Write()` routine. These two lines print on the same line. There is not a return line feed after printing. Lines 15 and 17 show each of these routines with the use of a parameter.

Printing Additional Information

In addition to printing text between quotation marks, you can also pass values to be printed within the text. Consider the following example:

```
System.Console.WriteLine("The following is a number: {0}", 456);
```

This prints

The following is a number: 456

As you can see, the `{0}` gets replaced with the value that follows the quoted text. The format is

```
System.Console.WriteLine("Text", value);
```

where `Text` is almost any text you want to display. The `{0}` is a placeholder for a value. The braces indicate that this is a placeholder. The `0` is an indicator for using the first item following the quotation marks. A comma separates the text from the value to be placed in the placeholder.

You can have more than one placeholder in a printout. Each placeholder is given the next sequential number. To print two values, use the following:

```
System.Console.WriteLine("Value 1 is {0} and value 2 is {1}", 123, "Brad");
```

This prints

Value 1 is 123 and value 2 is Brad

You will learn more about using these routines throughout this book.



Caution

The first placeholder is numbered `0` and not `1`.



Note

You can also see some weird text in line 15 of Listing 2.3. The `\n` on this line is not a mistake. This is an indicator that a newline should be started before printing the information that follows. You will learn more about this on Day 3, "Storing Information with Variables."

Summary

Today's lesson continued to help build a foundation that will be used to teach you C#.

Today you learned about some of the basic parts of a C# application. You learned that comments help make your programs easier to understand.

You also learned about the basic parts of a C# application, including whitespace, C# keywords, literals, and identifiers. Looking at an application, you saw how these parts are combined to create a complete listing. This included seeing a special identifier used as a starting point in an application—`Main()`.

After you examined a listing, you explored an overview of object-oriented programming, including the concepts of encapsulation, polymorphism, inheritance, and reuse.

2

Today's lesson concluded with some basic information on using the `System.Console.WriteLine()` and `System.Console.Write()` routines. You learned that these two routines are used to print information to the screen (console).

Q&A

Q What is the difference between component-based and object-oriented?

- A** C# has been referred to as component-based by some people. Component-based development can be considered an extension of object-oriented programming. A component is simply a standalone piece of code that performs a specific task. Component-based programming consists of creating lots of these standalone components that can be reused. You then link them to build applications.

Q What other languages are considered object-oriented?

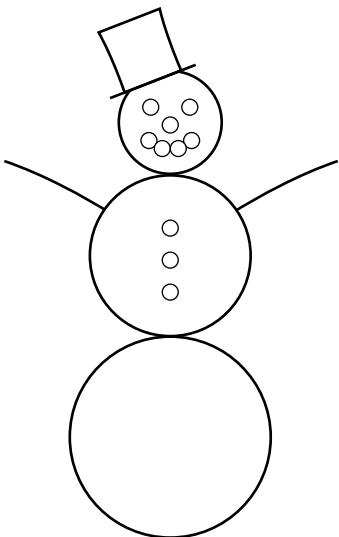
- A** Other languages that are considered object-oriented include C++, Java, and SmallTalk. Microsoft's Visual Basic.NET can also be used for object-oriented programming. There are other languages, but these are the most popular.

Q What is composition? Is it an object-oriented term?

- A** Many people confuse inheritance with composition, but they are different. With composition, one object is used within another object. Figure 2.1 is a composition of many circles. This is different from the sphere in the previous example. The sphere is not composed of a circle; it is an extension of the circle. To summarize the difference between composition and inheritance: Composition occurs when one class (object) has another within it. Inheritance occurs when one class (object) is an expansion of another.

FIGURE 2.1

A snowman made of circles.



Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. What are the three types of comments you can use in a C# program?
 2. How are each of the three types of comments entered into a C# program?
 3. What impact does whitespace have on a C# program?
 4. Which of the following are C# keywords?
`field, cast, as, object, throw, baseball, catch, football, fumble, basketball`
 5. What is a literal?
 6. Which is true:
Expressions are made up of statements.
Statements are made up of expressions.
Expressions and statements have nothing to do with each other.
7. What is an empty statement?
 8. What are the key concepts of object-oriented programming?

9. What is the difference between `WriteLine()` and `Write()`?
10. What is used as a placeholder when printing a value with `WriteLine()` or `Write()`?

Exercises

1. Enter and compile the following program (`listit.cs`). Remember, you don't enter the line numbers.

```
1: // ListIT.cs - program to print a listing with line numbers
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListIT
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
12:        {
13:
14:            int ctr=0;
15:            if (args.Length <= 0 )
16:            {
17:                Console.WriteLine("Format: ListIT filename");
18:                return;
19:            }
20:            else
21:            {
22:                FileStream f = new FileStream(args[0], FileMode.Open);
23:                try
24:                {
25:                    StreamReader t = new StreamReader(f);
26:                    string line;
27:                    while ((line = t.ReadLine()) != null)
28:                    {
29:                        ctr++;
30:                        Console.WriteLine("{0}: {1}", ctr, line);
31:                    }
32:                    f.Close();
33:                }
34:                finally
35:                {
36:                    f.Close();
37:                }
38:            }
39:        }
40:        catch (System.IO.FileNotFoundException)
41:        {
```

2

```
42:             Console.WriteLine ("ListIT could not find the file
{0}", args[0]);
43:         }
44:
45:         catch (Exception e)
46:         {
47:             Console.WriteLine("Exception: {0}\n\n", e);
48:         }
49:     }
50: }
```

2. Run the program entered in exercise 1. What happens when you run this program?

Run this program a second time. This time enter the following at the command line:

```
listit listit.cs
```

What happens this time?

3. Enter, compile, and run the following program. What does it do?

```
1: // ex0203.cs - Exercise 3 for Day 2
2: //-----
3:
4: class Exercise2
5: {
6:     public static void Main()
7:     {
8:         int x = 0;
9:
10:        for( x = 1; x <= 10; x++ )
11:        {
12:            System.Console.Write("{0:D3} ", x);
13:        }
14:    }
15: }
```

4. **BUG BUSTER:** The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
1: // bugbust.cs
2: //-----
3:
4: class bugbust
5: {
6:     public static void Main()
7:     {
8:         System.Console.WriteLine("\nA fun number is {1}", 123 );
9:     }
10: }
```

5. Write the line of code that prints your name on the screen.

WEEK 1

DAY 3

Storing Information with Variables

When you start writing programs, you are going to quickly find that you need to keep track of different types of information. This might be the tracking of your clients' names, the amounts of money in your bank accounts, or the ages of your favorite movie stars. To keep track of this information, your computer programs need a way to store the values. Today you

- Learn what is a variable
- Discover how to create variable names in C#
- Use different types of numeric variables
- Evaluate the differences and similarities between character and numeric values
- See how to declare and initialize variables

Variables

A *variable* is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the information stored there. For example, you could create a variable called `my_variable` that holds a number. You would be able to store different numbers in the `my_variable` variable.

You could also create variables to store information other than a simple number. You could create a variable called `BankAccount` to store a bank account number, a variable called `email` to store an email address, or a variable called `address` to store a person's mailing address. Regardless of what type of information will be stored, a variable is used to obtain its value.

Variable Names

To use variables in your C# programs, you must know how to create variable names. In C#, variable names must adhere to the following rules:

- The name can contain letters, digits, and the underscore character (_).
- The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended at the beginning of a name. An underscore is often used with special commands, and it's sometimes hard to read.
- Case matters (that is, upper- and lowercase letters). C# is case-sensitive; thus, the names `count` and `Count` refer to two different variables.
- C# keywords can't be used as variable names. Recall that a keyword is a word that is part of the C# language. (A complete list of the C# keywords can be found in Appendix B, "C# Keywords.")

The following list contains some examples of valid and invalid C# variable names:

Variable Name	Legality
<code>Percent</code>	Legal
<code>y2x5__w7h3</code>	Legal
<code>yearly_cost</code>	Legal
<code>_2010_tax</code>	Legal, but not advised
<code>checking#account</code>	Illegal; contains the illegal character #
<code>double</code>	Illegal; is a C keyword
<code>9byte</code>	Illegal; first character is a digit

Because C# is case-sensitive, the names `percent`, `PERCENT`, and `Percent` are considered three different variables. C# programmers commonly use only lowercase letters in variable names, although this isn't required. Often, programmers use mixed case as well. Using all-uppercase letters is usually reserved for the names of constants (which are covered later today).

Variables can have any name that fits the rules listed previously. For example, a program that calculates the area of a circle could store the value of the radius in a variable named `radius`. The variable name helps make its usage clear. You could also have created a variable named `x` or even `billy_gates`; it doesn't matter. Such a variable name, however, wouldn't be nearly as clear to someone else looking at the source code. Although it might take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.

Many naming conventions are used for variable names created from multiple words. Consider the variable name `circle_radius`. Using an underscore to separate words in a variable name makes it easy to interpret. Another style is called *Pascal notation*. Instead of using spaces, the first letter of each word is capitalized. Instead of `circle_radius`, the variable would be named `CircleRadius`. Yet another notation that is growing in popularity is *Camel notation*. Camel notation is like Pascal notation, except the first letter of the variable name is also lower case. A special form of Camel notation is called Hungarian notation. With Hungarian notation, you also include information in the name of the variable—such as whether it is numeric, has a decimal value, or is text—that helps to identify the type of information being stored. The underscore is used in this book because it's easier for most people to read. You should decide which style you want to adopt.

3

Do	DON'T
<p>DO use variable names that are descriptive.</p> <p>DO adopt and stick with a style for naming your variables.</p>	<p>DON'T name your variables with all capital letters unnecessarily.</p>



Note

C# supports a Unicode character set, which means that letters from any language can be stored and used. You can also use any Unicode character to name your variables.

Using Variables

Before you can use a variable in a C# program, you must declare it. A variable declaration tells the compiler the name of the variable and the type of information the variable will be used to store. If your program attempts to use a variable that hasn't been declared, the compiler will generate an error message.

Declaring a variable also enables the computer to set aside memory for it. By identifying the specific type of information that will be stored in a variable, you gain the best performance and avoid wasting memory.

Declaring a Variable

A variable declaration has the following form:

```
typename varname;
```

typename specifies the variable type. In the following sections you will learn about the types of variables that are available in C#. *varname* is the name of the variable. To declare a variable that can hold a standard numeric integer, you use the following line of code:

```
int my_number;
```

The name of the variable declared is *my_number*. The data type of the variable is *int*. As you will learn in the following section, the type *int* is used to declare integer variables, which is perfect for this example!

You can also declare multiple variables of the same type on one line by separating the variable names with commas. This enables you to be more concise in your listings.

Consider the following line:

```
int count, number, start;
```

This line declares three variables: *count*, *number*, and *start*. Each of these variables is type *int*, which is for integers.



Note

Although declaring multiple variables on the same line can be more concise, I don't recommend that you always do this. There are times when it is easier to read and follow your code by using multiple declarations. There will be no noticeable performance loss by doing separate declarations.

Assigning Values to Variables

Now that you know how to declare a variable it is important to learn how to store values. After all, the purpose of a variable is to store information!

The format for storing information in a variable is

```
varname = value;
```

You have already seen that `varname` is the name of the variable. `value` is the value that will be stored in the variable. For example, to store the number 5 in the variable, `my_variable`, you enter the following:

```
my_variable = 5;
```

To change the value, you simply reassign a new value:

```
my_variable = 1010;
```

Listing 3.1 illustrates assigning values to a variable. It also shows that you can overwrite a value.

3

LISTING 3.1 var_values.cs—Assigning Values to a Variable

```
1: // var_values.cs - A listing to assign and print the value
2: // of a variable
3: //-----
4:
5: using System;
6:
7: class var_values
8: {
9:     public static void Main()
10:    {
11:        // declare my_variable
12:        int my_variable;
13:
14:        // assign a value to my_variable
15:        my_variable = 5;
16:        Console.WriteLine("\nmy_variable contains the value {0}",
17:                         my_variable);
18:
19:        // assign a new value to my_variable
20:        my_variable = 1010;
21:        Console.WriteLine("\nmy_variable contains the value {0}",
22:                         my_variable);
23:    }
24: }
```

OUTPUT

```
my_variable contains the value 5  
my_variable contains the value 1010
```

ANALYSIS

Enter this listing into your editor, compile it, and execute it. If you need a refresher on how to do this, refer to Day 1, “Getting Started with C#.” The first three lines of this listing are comments. Lines 11, 14, and 18 also contain comments. Remember that comments provide information; the compiler will ignore them. Line 5 includes the `System` namespace which you need to do things such as writing information. Line 7 declares the class that will be your program (`var_values`). Line 9 declares the entry point for your program, the `Main()` function. Remember, `Main()` has to be capitalized or you’ll get an error!

Line 12 is the beginning point of today’s lesson. Line 12 declares a variable called `my_variable` of type integer (`int`). After this line has executed, the computer knows that a variable called `my_variable` exists and it enables you to use it. In line 15 you use this variable and assign the value of 5 to `my_variable`. In line 16 you use `Console.WriteLine` to display the value of `my_variable`. As you can see in the output, the value 5—which was assigned in line 15—is displayed. In line 19 you change the value of `my_variable` from 5 to `1010`. You see that this new assignment worked because the call to `Console.WriteLine` in line 20 prints the new value `1010` instead of 5.

After you assign the value `1010` to `my_variable` in line 19, the value of 5 is gone. From that point, the program no longer knows that 5 ever existed.

**Note**

You must declare a variable before you can use it. A variable can, however, be declared at almost any place within a listing.

Setting Initial Values in Variables

You might be wondering whether you can assign a value to a variable at the same time you declare it. Yes, you definitely can. In fact, it is good practice to make sure you always initialize a variable at the time you declare it. To initialize `my_variable` to the value of 8 when you declare it, you combine what you’ve done before:

```
int my_variable = 8;
```

Any variable can be initialized when being declared using the following structure:

```
typename varname = value;
```

You can also declare multiple variables on the same line and assign values to each of them:

```
int my_variable = 8, your_variable = 1000;
```

This line declares two integer variables called `my_variable` and `your_variable`. `my_variable` is assigned the value of 8 and `your_variable` is assigned 1000. Notice that these declarations are separated by a comma and that the statement ends with the standard semicolon. Listing 3.2 shows this statement in action.

LISTING 3.2 multi_variables.cs—Assigning More than One Variable

```
1: // multi_variables.cs
2: // A listing to assign values to more than one variable.
3: //-----
4:
5: using System;
6:
7: class multi_variables
8: {
9:     public static void Main()
10:    {
11:        // declare the variables
12:        int my_variable = 8, your_variable = 1000;
13:
14:        // print the original value of my_variable
15:        Console.WriteLine("my_variable was assigned the value {0}",
16:                         my_variable);
17:
18:        // assign a value to my_variable
19:        my_variable = 5;
20:
21:        // print their values
22:        Console.WriteLine("\nmy_variable contains the value {0}",
23:                         my_variable);
24:        Console.WriteLine("\nyour_variable contains the value {0}",
25:                         your_variable);
26:    }
27: }
```

3

OUTPUT

```
my_variable was assigned the value 8
my_variable contains the value 5
your_variable contains the value 1000
```

ANALYSIS

This listing declares and initializes two variables on line 12. The variable, `my_variable`, is initialized to the value of 8 and the variable, `your_variable`, is initialized to the value of 1000. Line 15 prints the value of `my_variable` so you can see what it contains. Looking at the output, you see that it contains the value 8, which was just assigned.

In line 18, the value of 5 is assigned to `my_variable`. Lines 21 and 22 print the final values of the two variables. The value of `my_variable` is printed as 5. The original value of 8 is gone forever! `your_variable` still contains its original value of 1000.

Using Uninitialized Variables

What happens if you try to use a variable without initializing it? Consider the following:

```
int blank_variable;
Console.WriteLine("\nmy_variable contains the value {0}", blank_variable);
```

In this snippet of code, `blank_variable` is printed in the second line. What is the value of `blank_variable`? This variable was declared in the first line, but it was not initialized to any value. You'll never know what the value of `blank_variable` is because the compiler will not create the program. Listing 3.3 proves this.

LISTING 3.3 blank.cs—Using an Uninitialized Variable

```
1: // blank.cs - Using unassigned variables.
2: // This listing causes an error!!
3: //-----
4:
5: using System;
6:
7: class blank
8: {
9:     public static void Main()
10:    {
11:        int blank_variable;
12:
13:        Console.WriteLine("\nmy_variable contains the value {0}",
14:                         blank_variable);
15:    }
16: }
```

ANALYSIS

This program will not compile. Rather, the compiler will give you the following error:

```
blank.cs(13,67): error CS0165: Use of unassigned local variable 'blank_variable'
```

C# will not let you use an uninitialized variable.

**Note**

In other languages, such as C and C++, this listing would compile. The value printed for the `blank_variable` in these other languages would be garbage. C# prevents this type of error from occurring.

Understanding Your Computer's Memory

If you already know how a computer's memory operates, you can skip this section. If you're not sure, read on. This information is helpful to understanding programming.

What is your computer's RAM used for? It has several uses, but only data storage need concern you as a programmer. Data is the information with which your C# program works. Whether your program is maintaining a contact list, monitoring the stock market, keeping a budget, or tracking the price of snickerdoodles, the information (names, stock prices, expense amounts, or prices) is kept within variables in your computer's RAM when it is being used by your running program.

A computer uses random access memory (RAM) to store information while it is operating. RAM is located in integrated circuits, or chips, inside your computer. RAM is volatile, which means that it is erased and replaced with new information as often as needed. Being volatile also means that RAM "remembers" only while the computer is turned on and loses its information when you turn the computer off.

3

A *byte* is the fundamental unit of computer data storage. Each computer has a certain amount of RAM installed. The amount of RAM in a system is usually specified in megabytes (MB), such as 32MB, 64MB, 128MB or more. One megabyte of memory is 1,024 kilobytes (KB). One kilobyte of memory consists of 1,024 bytes. Thus, a system with 8MB of memory actually has $8 \times 1,024\text{KB}$, or 8,192KB of RAM. This is $8,192\text{KB} \times 1,024$ bytes for a total of 8,388,608 bytes of RAM. Table 3.1 provides you with an idea of how many bytes it takes to store certain kinds of data.

TABLE 3.1 Minimum Memory Space Generally Required to Store Data

Data	Bytes Required
The letter x	2
The number 500	2
The number 241.105	4
The phrase <i>Teach Yourself C#</i>	34
One typewritten page	Approximately 4,000

The RAM in your computer is organized sequentially, one byte following another. Each byte of memory has a unique address by which it is identified—an address that also distinguishes it from all other bytes in memory. Addresses are assigned to memory locations in order, starting at 0 and increasing to the system limit. For now, you don't need to worry about addresses; it's all handled automatically.

Now that you understand a little about the nuts and bolts of memory storage, you can get back to C# programming and how C# uses memory to store information.

C# Data Types

You know how to declare, initialize, and change the values of variables; it is important that you know the data types you can use. You learned earlier that you have to declare the data type when you declare a variable. You've seen that the `int` keyword declares variables that can hold integers. An integer is simply a whole number that doesn't contain a fractional or decimal portion. The variables you've declared to this point hold only integers. What if you want to store other types of data, such as decimals or characters?

Numeric Variable Types

C# provides several different types of numeric variables. You need different types of variables, because different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them. Small integers (for example, 1, 199, and -8) require less memory to store, and your computer can perform mathematical operations (addition, multiplication, and so on) with such numbers very quickly. In contrast, large integers and values with decimal points require more storage space and more time for mathematical operations. By using the appropriate variable types, you ensure that your program runs as efficiently as possible.

The following sections break the different numeric data types into four categories:

- Integral
- Floating-point
- Decimal
- Boolean

Earlier in today's lesson, you learned that variables are stored in memory. Additionally, you learned that different types of information required different amounts of memory. The amount of memory used to store a variable is based on its data type. Listing 3.4 is a program that contains code beyond what you know right now; however, it provides you with the amount of information needed to store some of the different C# data types.

NEW TERM You must include extra information for the compiler when you compile this listing. This extra information, referred to as a *flag* to the compiler, can be included on the command line. Specifically, you need to add the `/unsafe` flag as shown:

```
csc /unsafe sizes.cs
```

If you are using an integrated development environment, you need to set the `unsafe` option as instructed by its documentation.

LISTING 3.4 sizes.cs—Memory Requirements for Data Types

```
1: // sizes.cs--Program to tell the size of the C# variable types
2: //-----
3:
4: using System;
5:
6: class sizes
7: {
8:     unsafe public static void Main()
9:     {
10:         Console.WriteLine( "\nA byte      is {0} byte(s)", sizeof( byte ) );
11:         Console.WriteLine( "An sbyte    is {0} byte(s)", sizeof( sbyte ) );
12:         Console.WriteLine( "A char      is {0} byte(s)", sizeof( char ) );
13:         Console.WriteLine( "\nA short     is {0} byte(s)", sizeof( short ) );
14:         Console.WriteLine( "An ushort   is {0} byte(s)", sizeof( ushort ) );
15:         Console.WriteLine( "\nAn int      is {0} byte(s)", sizeof( int ) );
16:         Console.WriteLine( "An uint      is {0} byte(s)", sizeof( uint ) );
17:         Console.WriteLine( "\nA long      is {0} byte(s)", sizeof( long ) );
18:         Console.WriteLine( "An ulong     is {0} byte(s)", sizeof( ulong ) );
19:         Console.WriteLine( "\nA float      is {0} byte(s)", sizeof( float ) );
20:         Console.WriteLine( "A double     is {0} byte(s)", sizeof( double ) );
21:         Console.WriteLine( "\nA decimal   is {0} byte(s)", sizeof( decimal )
22:             );
22:         Console.WriteLine( "\nA boolean   is {0} byte(s)", sizeof( bool ) );
23:     }
24: }
```

3

OUTPUT

```
A byte      is 1 byte(s)
An sbyte    is 1 byte(s)
A char      is 2 byte(s)

A short     is 2 byte(s)
An ushort   is 2 byte(s)

An int      is 4 byte(s)
An uint      is 4 byte(s)

A long      is 8 byte(s)
An ulong     is 8 byte(s)

A float      is 4 byte(s)
A double     is 8 byte(s)

A decimal   is 16 byte(s)

A boolean   is 1 byte(s)
```

ANALYSIS Although you haven't learned all the data types yet, I believed it valuable to present this listing here. As you go through the following sections, refer to this listing and its output.

This listing uses a C# keyword called `sizeof`. The `sizeof` keyword tells you the size of a variable. In this listing, `sizeof` is used to show the size of the different data types. For example, to determine the size of an `int`, you can use:

```
sizeof(int)
```

If you had declared a variable called `x`, you could determine its size—which would actually be the size of its data type—by using the following code:

```
sizeof(x)
```

Looking at the output of Listing 3.4, you see that you have been given the number of bytes that are required to store each of the C# data types. For an `int`, you need 4 bytes of storage. For a `short` you need 2. The amount of memory used determines how big or small a number can be that is stored. You'll learn more about this in the following sections.

The `sizeof` keyword is not one that you will use very often; however, it is useful for illustrating the points in today's lesson. The `sizeof` keyword taps into memory to determine the size of the variable or data type. With C#, you avoid tapping directly into memory. In line 8, an extra keyword is added—`unsafe`. If you don't include the `unsafe` keyword, you get an error when you compile this program. For now, understand that the reason `unsafe` is added is because the `sizeof` keyword works directly with memory.



Caution

The C# keyword `sizeof` can be used; however, you should generally avoid it. The `sizeof` keyword sometimes accesses memory directly to find out the size. Accessing memory directly is something to be avoided in pure C# programs.

Integral Data Types

Until this point, you have been using one of the integral data types—`int`. Integral data types store integers. Recall that an integer is basically any numeric value that does not include a decimal or a fractional value. The numbers 1, 1,000, 56 trillion, and -534 are integral values.

C# provides nine integral data types, including the following:

- Integers (`int` and `uint`)
- Shorts (`short` and `ushort`)

- Longs (`long` and `ulong`)
- Bytes (`byte` and `sbyte`)
- Characters (`char`)

Integers

As you saw in Listing 3.4, an integer is stored in 4 bytes of memory. This includes both the `int` and `uint` data types. The `int` data type has been used in many of the programs you have seen so far. Although you might not have known it, this data type cannot store just any number. Rather, it can store any signed whole number that can be represented in 4 bytes or 32 bits—any number between -2,147,483,648 and 2,147,483,647.

A variable of type `int` is signed, which means it can be positive or negative. Technically, 4 bytes can hold a number as big as 4,294,967,295; however, when you take away one of the 32 bits to keep track of positive or negative, you can go only to 2,147,483,647. You can, however, also go to -2,147,483,648.

3

As you learned earlier, information is stored in units called bytes. A byte is actually composed of 8 bits. A *bit* is the most basic unit of storage in a computer. A bit can have one of two values—0 or 1. Using bits and the binary math system, you can store numbers in multiple bits. In Appendix C, “Working with Number Systems,” you can learn the details of binary math.

If you want to use a type `int` to go higher, you can make it `unsigned`. An `unsigned` number can only be positive. The benefit should be obvious. The `uint` data type declares an `unsigned` integer. The net result is that a `uint` can store a value from 0 to 4,294,967,295.

What happens if you try to store a number that is too big? What about storing a number with a decimal point into an `int` or `uint`? What happens if you try to store a negative number into an `uint`? Listing 3.5 answers all three questions.

LISTING 3.5 int_conv.cs—Doing Bad Things

```
1: // int_conv.cs
2: // storing bad values. Program generates errors and won't compile.
3: //-----
4:
5: using System;
6:
7: class int_conv
8: {
9:     public static void Main()
```

LISTING 3.5 continued

```
10:      {
11:          int val1, val2;      // declare two integers
12:          uint pos_val;       // declare an unsigned int
13:
14:          val1 = 1.5;
15:          val2 = 9876543210;
16:          pos_val = -123;
17:
18:          Console.WriteLine( "val1 is {0}", val1);
19:          Console.WriteLine( "val2 is {0}", val2);
20:          Console.WriteLine( "pos_val is {0}", pos_val);
21:      }
22: }
```

OUTPUT

```
int_conv.cs(14,15): error CS0029: Cannot implicitly convert type
  ↪'double' to 'int'
int_conv.cs(15,15): error CS0029: Cannot implicitly convert type 'long'
  ↪to 'int'
int_conv.cs(16,18): error CS0031: Constant value '-123' cannot be
  ↪converted to a 'uint'
```



This program gives compiler errors.

ANALYSIS

This program will not compile. As you can see, the compiler catches all three problems that were questioned. In line 14 you try to put a number with a decimal point into an integer. In line 15 you try to put a number that is too big into an integer. Remember, the highest number that can go into an `int` is 2,147,483,647. Finally, in line 16, you try to put a negative number into an unsigned integer (`uint`). As the output shows, the compiler catches each of these errors and prevents the program from being created.

Shorts

The `int` and `uint` data types used 4 bytes of memory for each variable declared. There are a number of times when you don't need to store numbers that are that big. For example, you don't need big numbers to keep track of the day of the week (numbers 1 to 7), to store a person's age, or to track the temperature to bake a cake.

When you want to store a whole number and you want to save some memory, you can use `short` and `ushort`. A `short`, like an `int`, stores a whole number. Unlike an `int`, it is

only 2 bytes instead of 4. If you look at the output from Listing 3.4, you see that `sizeof` returned 2 bytes for both `short` and `ushort`. If you are storing both positive and negative numbers, you'll want to use `short`. If you are storing only positive and you want to use the extra room, you'll want to use `ushort`. The values that can be stored in a `short` are from -32,768 to 32,767. If you use a `ushort`, you can store whole numbers from 0 to 65535.

Longs

If `int` and `uint` are not big enough for what you want to store, there is another data type to use—`long`. As with `short` and `int`, there is also an unsigned version of the `long` data type called `ulong`. Looking at the output from Listing 3.4, you can see that `long` and `ulong` each use 8 bytes of memory. This gives them the capability of storing very large numbers. A `long` can store numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. An `ulong` can store a number from 0 to 18,446,744,073,709,551,615.

3

Bytes

As you have seen, you can store whole numbers in data types that take 2, 4, or 8 bytes of memory. For those times when your needs are very small, you can store a whole number in a single byte. To keep things simple, the data type that uses a single byte of memory for storage is called a `byte`! As with the previous integers, there is both a signed version, `sbyte`, and an unsigned version, `byte`. A `sbyte` can store a number from -128 to 127. An unsigned byte can store a number from 0 to 255.

Characters

In addition to numbers, you will often want to store characters. Characters are letters, such as *A*, *B*, or *C*, or even extended characters such as the smiley face. Additional characters that you might want to store are characters such as \$, %, or *. You might even want to store foreign characters.

A computer does not recognize characters. It can recognize only numbers. To get around this, all characters are stored as numeric values. To make sure that everyone uses the same values, a standard was created called Unicode. Within Unicode, each character and symbol is represented by a single whole number. This is why the character data type is considered an integral type.

To know that numbers should be used as characters, you use the data type `char`. A `char` is a number stored in 2 bytes of memory that is interpreted as a character. Listing 3.6 presents a program that uses `char` values.

LISTING 3.6 chars.cs—Working with Characters

```
1: // chars.cs
2: // A listing to print out a number of characters and their numbers
3: //-----
4:
5: using System;
6:
7: class chars
8: {
9:     public static void Main()
10:    {
11:        int ctr;
12:        char ch;
13:
14:        Console.WriteLine("\nNumber      Value\n");
15:
16:        for( ctr = 60; ctr <= 95; ctr = ctr + 1)
17:        {
18:            ch = (char) ctr;
19:            Console.WriteLine( "{0} is {1}", ctr, ch);
20:        }
21:    }
22: }
```

OUTPUT

Number	Value
60	is <
61	is =
62	is >
63	is ?
64	is @
65	is A
66	is B
67	is C
68	is D
69	is E
70	is F
71	is G
72	is H
73	is I
74	is J
75	is K
76	is L
77	is M
78	is N
79	is O
80	is P
81	is Q
82	is R
83	is S

60	is <
61	is =
62	is >
63	is ?
64	is @
65	is A
66	is B
67	is C
68	is D
69	is E
70	is F
71	is G
72	is H
73	is I
74	is J
75	is K
76	is L
77	is M
78	is N
79	is O
80	is P
81	is Q
82	is R
83	is S

```
84 is T  
85 is U  
86 is V  
87 is W  
88 is X  
89 is Y  
90 is Z  
91 is [  
92 is \  
93 is ]  
94 is ^  
95 is _
```

ANALYSIS This listing displays a range of numeric values and their character equivalents. In line 11 an integer is declared called `ctr`. This variable is used to cycle through a number of integers. Line 12 declares a character variable called `ch`. Line 14 prints headings for the information that will be displayed.

3

Line 16 contains something new. For now, don't worry about fully understanding this line of code. On Day 5, "Control Statements," you learn all the glorious details. For now know that this line sets the value of `ctr` to 60. It then runs lines 18 and 19 before adding 1 to the value of `ctr`. It keeps doing this until `ctr` is no longer less than or equal to 95. The end result is that lines 18 and 19 are run using the `ctr` with the value of 60, then 61, then 62, and on and on until `ctr` is 95.

Line 18 sets the value of `ctr` (first 60) and places it into the character variable, `ch`. Because `ctr` is an integer, you have to tell the computer to convert the integer to a character, which the `(char)` statement does. You'll learn more about this later.

Line 19 prints the values stored in `ctr` and `ch`. As you can see, the integer `ctr` prints as a number. The value of `ch`, however, does not print as a number; it prints as a character. As you can see from the output of this listing, the character A is represented by the value 65. The value of 66 is the same as the character B.



Note

A computer actually recognizes only 1s and 0s (within bits). It recognizes these as on or off values (or positive charges versus negative charges). A binary number system is one that uses 1s and 0s to represent its numbers. Appendix C explains the binary number system.

Character Literals

How can you assign a character to a `char` variable? You place the character between single quotes. For example, to assign the letter a to the variable `my_char`, you use the following:

```
my_char = 'a';
```

In addition to assigning regular characters, there are also several extended characters you will most likely want to use. You have actually been using one extended character in a number of your listings. The `\n` that you've been using in your listings is an extended character. This prints a newline character. Table 3.2 contains some of the most common characters you might want to use. Listing 3.7 shows some of these special characters in action.

TABLE 3.2 Extended Characters

Characters	Meaning
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\"</code>	Backslash
<code>\'</code>	Single quote
<code>\"</code>	Double quote



Note

The extended characters in Table 3.2 are often called *escape characters* because the slash “escapes” from the regular text and indicates that the following character is special (or extended).

LISTING 3.7 chars_table.cs—The Special Characters

```
1: // chars_table.cs
2: //-----
3:
4: using System;
5:
6: class chars_table
7: {
8:     public static void Main()
9:     {
10:         char ch1 = 'Z';
11:         char ch2 = 'x';
12:
13:         Console.WriteLine("This is the first line of text");
14:         Console.WriteLine("\n\n\nSkipped three lines");
15:         Console.WriteLine("one\ttwo\tthree <-tabbed");
16:         Console.WriteLine(" A quote: \' \ndouble quote: \"");
17:         Console.WriteLine("\n ch1 = {0}    ch2 = {1}", ch1, ch2);
18:     }
19: }
```

OUTPUT

This is the first line of text

```
Skipped three lines
one      two      three <-tabbed
A quote: '
double quote: "

ch1 = Z    ch2 = x
```

ANALYSIS

ALYSIS This listing illustrates two concepts. First, in line 10 and 11 you see how a character can be assigned to a variable of type `char`. It is as simple as including the character in single quotes. In lines 13 to 17, you see how to use the extended characters. There is nothing special about line 13. Line 14 prints three newlines followed by some text. Line 15 prints one, two, and three, separated by tabs. Line 16 displays a single quote and a double quote. Notice that there are two double quotes in a row at the end of this line. Finally, line 17 prints the values of `ch1` and `ch2`.

3

Floating Point

Not all numbers are whole numbers. For those times when you need to use numbers that might have decimals, you need to use different data types. As with storing whole numbers, there are different data types you can use, depending on the size of the numbers you are using and the amount of memory you want to use. The two primary types are `float` and `double`.

float

A `float` is a data type for storing numbers with decimal places. For example, in calculating the circumference or area of a circle, you often end up with a result that is not a whole number. Any time you need to store a number such as 1.23 or 3.1459, you need a nonintegral data type.

The `float` data type stores numbers in 4 bytes of memory. As such, it can store a number from approximately 1.5×10^{-45} to 3.4×10^{38} .



Note



Caution

A `float` can retain only about 7 digits of precision, which means it is not uncommon for a `float` to be off by a fraction. For example, subtracting 9.90 from 10.00 might result in a number different from .10. It might result in a number closer to .09999999. Generally such rounding errors are not noticeable.

double

Variables of type `double` are stored in 8 bytes of memory. This means they can be much bigger than a `float`. A `double` can generally be from 5.0×10^{-324} to 1.7×10^{308} . The precision of a `double` is generally from 15 to 16 digits.



Note

C# supports the 4-byte precision (32 bits) and 8-byte precision (64 bits) of the IEEE 754 format, so certain mathematical functions return specific values. If you divide a number by 0, the result is infinity (either positive or negative). If you divide 0 by 0, you get a *Not-a-Number* value. Finally, 0 can be both positive and negative. For more on this, check your C# documentation.

Decimal

C# provides another data type that can be used to store special decimal numbers. This is the `decimal` data type. This data type was created for storing numbers with greater precision. When you store numbers in a `float` or `double`, you can get rounding errors. For example, storing the result of subtracting 9.90 from 10.00 in a `double` could result in the string `0.099999999999999645` instead of `.10`. If this math is done with `decimal` values, the `.10` is stored.



Tip

If you are calculating monetary values or doing financial calculations where precision is important, you should use a `decimal` instead of a `float` or a `double`.

A `decimal` number uses 16 bytes to store numbers. Unlike the other data types, there is not an unsigned version of `decimal`. A `decimal` variable can store a number from 1.0×10^{-28} to approximately 7.9×10^{28} . It can do this while maintaining precision to 28 places.

Boolean

The last of the simple data types is the Boolean. Sometimes you need to know whether something is on or off, true or false, yes or no. Boolean numbers are generally set to one of two values: 0 or 1.

C# has a Boolean data type called a `bool`. As you can see in Listing 3.4, a `bool` is stored in 1 byte of memory. The value of a `bool` is either `true` or `false`, which are C# keywords. This means you can actually store `true` and `false` in a data type of `bool`.



Caution

"Yes," "no," "on," and "off" are not keywords in C#. This means you cannot set a Boolean variable to these values. Instead, you must use `true` or `false`.

3

Checking Versus Unchecking

Earlier in today's lesson you learned that if you put a number that is too big into a variable, an error is produced. There are times when you might not want an error produced. In those cases, you can have the compiler avoid checking the code. This is done with the `unchecked` keyword. Listing 3.8 illustrates this.

LISTING 3.8 unchecked.cs—Marking Code as Unchecked

```
1: // unchecked.cs
2: //-----
3:
4: using System;
5:
6: class sizes
7: {
8:     public static void Main()
9:     {
10:         int val1 = 2147483647;
11:         int val2;
12:
13:         unchecked
14:         {
15:             val2 = val1 + 1;
16:         }
17:
18:         Console.WriteLine( "val1 is {0}", val1);
19:         Console.WriteLine( "val2 is {0}", val2);
20:     }
21: }
```

OUTPUT

```
val1 is 2147483647  
val2 is -2147483648
```

ANALYSIS

This listing uses `unchecked` in line 13. The brackets on line 14 and 16 enclose the area to be unchecked. When you compile this listing, you do not get any errors. When you run the listing, you get what might seem like a weird result. The number 2,147,483,647 is the largest number that a signed `int` variable can hold. As you see in line 10, this maximum value has been assigned to `var1`. In line 15, the `unchecked` line, 1 is added to what is already the largest value `var1` can hold. Because this line is `unchecked`, the program continues to operate. The result is that the value stored in `var1` rolls to the most negative number.

This operation is similar to the way an odometer works in a car. When the mileage gets to the maximum, such as 999,999, adding 1 more mile (or kilometer) sets the odometer to 000,000. It isn't a new car with no miles, it is simply a car that no longer has a valid value on its odometer. Rather than rolling to 0, a variable is going to roll to the lowest value it can store. In this listing, that value is -2,147,483,648.

Change line 13 to the following and recompile and run the listing:

```
13:      checked
```

The program compiled, but will it run? Executing the program causes an error. If you are asked to run your debugger, you'll want to say no. The error you get will be similar to the following:

```
Exception occurred: System.OverflowException: An exception of type  
↳System.OverflowException was thrown.  
  at sizes.Main()
```

On later days, you'll see how to deal with this error in your program. For now, you should keep in mind that if you believe there is a chance to put an invalid value into a variable, you should force checking to occur.

Data Types Simpler than .NET

The C# data types covered so far are considered simple data types. The simple data types are `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `bool`, and `decimal`. On Day 1 and Day 2, “Understanding C# Programs,” you learned that C# programs execute on the Common Language Runtime (CLR). Each of these data types corresponds directly to a data type that the CLR uses. Each of these types is considered simple because there is a direct relationship to the types available in the CLR and thus in the .NET Framework. Table 3.3 presents the .NET equivalent of the C# data types.

TABLE 3.3 C# and .NET Data Types

C# Data Type	.NET Data Type
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

If you want to declare an integer using the .NET equivalent declaration—even though there is no good reason to do so—you use the following:

```
System.Int32 my_variable = 5;
```

As you can see, `System.Int32` is much more complicated than simply using `int`. Listing 3.9 shows the use of the .NET data types.

LISTING 3.9 net_vars.cs—Using the .NET Data Types

```
1: // net_vars
2: // Using a .NET data declaration
3: //-----
4:
5: using System;
6:
7: class net_vars
8: {
9:     public static void Main()
10:    {
11:
12:        System.Int32 my_variable = 4;
13:        System.Double PI = 3.1459;
14:    }
}
```

LISTING 3.9 continued

```
15:         Console.WriteLine("\nmy_variable is {0}", my_variable );
16:         Console.WriteLine("\nPI is {0}", PI );
17:     }
18: }
```

OUTPUT

```
my_variable is 4
```

```
PI is 3.1459
```

ANALYSIS

Lines 12 and 13 declare an `int` and a `double`. Lines 15 and 16 print these values. This listing operates like those you've seen earlier, except it uses the .NET data types.

In your C# programs, you should use the simple data types rather than the .NET types. All the functionality that the .NET types have is available to you in the simpler commands that C# provides. You should, however, understand that the simple C# data types translate to .NET equivalents. You'll find that all other programming languages that work with the Microsoft .NET types also have data types that translate to these .NET types.

 **Note**

The *Common Type System (CTS)* is a set of rules that data types within the CLR must adhere to. The simple data types within C# adhere to these rules, as do the .NET data types. If a language follows the CTS in creating its data types, the data created and stored should be compatible with other programming languages that also follow the CTS.

Literals Versus Variables

NEW TERM

In the examples you've looked at so far, you have seen a lot of numbers and values used that were not variables. Often, you will want to type a number or value into your source code. A *literal value* stands on its own within the source code. For example, in the following lines of code, the number `10` and the value `"Bob is a fish"` are literal values. As you can see, literal values can be put into variables.

```
int x = 10;
myStringValue = "Bob is a fish";
```

Numeric Literals

In many of the examples, you have used numeric literals. By default, a numeric literal is either an `int` or a `double`. It is an `int` if it is a whole number, and it is a `double` if it is a floating-point number. For example, consider the following:

```
nbr = 100;
```

In this example, 100 is a numeric literal. By default, it is considered to be of type `int`, regardless of what data type the `nbr` variable is. Now consider the following:

```
nbr = 99.9;
```

In this example, 99.9 is also a numeric literal; however, it is of type `double` by default. Again, this is regardless of the data type that `nbr` is. This is true even though 99.9 could be stored in a type `float`. In the following line of code, is 100. an `int` or a `double`?

```
x = 100.;
```

This is a tough one. If you guessed `int`, you are wrong. Because there is a decimal included with the 100, it is a `double`.

3

Integer Literal Defaults

When you use an integer value, it is put into an `int`, `uint`, `long`, or `ulong` depending on its size. If it will fit in an `int` or `uint`, it will be. If not, it will be put into a `long` or `ulong`. If you want to specify the data type of the literal, you can use a suffix on the literal. For example, to use the number 10 as a literal `long` value(signed or unsigned), you write it like the following:

```
10L;
```

You can make an `unsigned` value by using a `u` or a `U`. If you want an `unsigned` literal `long` value, you can combine the two suffixes: `ul`.



Note

The Microsoft C# compiler gives you a warning if you use a lowercase `l` to declare a `long` value literal. The compiler provides this warning to help you be aware that it is easy to mistake a lowercase `l` with the number 1.

Floating-Point Literal Defaults

As stated earlier, by default, a decimal value literal is a `double`. To declare a literal that is of type `float`, you include an `f` or `F` after the number. For example, to assign the number 4.4 to a `float` variable, `my_float`, you use the following:

```
my_float = 4.4f;
```

To declare a literal of type `decimal`, you use a suffix of `m` or `M`. For example, the following line declares `my_decimal` to be equal to the decimal number 1.32.

```
my_decimal = 1.32m;
```

Boolean Literals (`true` and `false`)

Boolean literals have already been covered. The values `true` and `false` are literal. They also happen to be keywords.

String Literals

When you put characters together, they make words, phrases, and sentences. In programming parlance, a group of characters is called a *string*. A string can be identified because it is contained between a set of double quotes. You have actually been using strings in the examples so far. For example, the `Console.WriteLine` routine uses a string. A string literal is any set of characters between double quotes. The following are examples of strings:

`"Hello, World!"`

`"My Name is Bradley"`

`"1234567890"`

Because these numbers are between quotation marks, the last example is treated as a string literal rather than as a numeric literal.



You can use any of the special characters from Table 3.3 inside a string.

Constants

In addition to using literals, there are times when you want to put a value in a variable and freeze it. For example, if you declare a variable called `PI` and you set it to `3.1459`, you want it to stay `3.1459`. There is no reason to change it. Additionally, you want to prevent people from changing it.

To declare a variable to hold a constant value, you use the `const` keyword. For example, to declare `PI` as stated, you use the following:

```
const float PI = 3.1459;
```

You can use `PI` in a program; however, you will never be able to change its value. The `const` keyword freezes its contents. You can use the `const` keyword on any variable of any data type.



To help make it easy to identify constants, you can enter their names in all capital letters. This makes it easy to identify the fact that the variable is a constant.

Reference Types

To this point, you have seen a number of different data types. C# offers two primary ways of storing information: by value (byval) and by reference (byref). The basic data types that you have learned about store information by value.

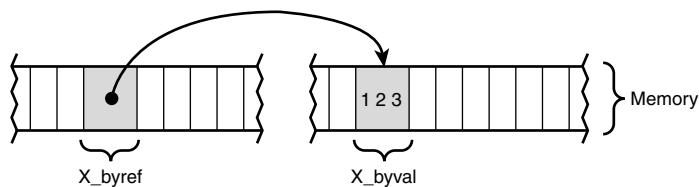
When a variable stores information by value, the variable contains the actual information. For example, when you store 123 in an integer variable called *x*, the value of *x* is 123. The variable *x* actually contains the value 123.

3

Storing information by reference is a little more complicated. If a variable stores by reference, rather than storing the information in itself, it stores the location of the information. In other words, it stores a reference to the information. For example, if *x* is a “by reference” variable, it contains information on where the value 123 is located. It does not store the value 123. Figure 3.1 illustrates the difference.

FIGURE 3.1

By reference versus by value.



The data types used by C# that store by reference are

- Classes
- Strings
- Interfaces
- Arrays
- Delegates

Each of these data types is covered in detail throughout the rest of this book.

Summary

In today's lesson, you learned how the computer stores information. You focused on the data types that store data by value, including `int`, `uint`, `long`, `ulong`, `bool`, `char`, `short`, `ushort`, `float`, `double`, `decimal`, `byte`, and `ubyte`. In addition to learning about the data types, you learned how to name and create variables. You also learned the basics of setting values in these variables, including the use of literals. Table 3.4 lists the data types and information about them.

TABLE 3.4 C# Data Types

C# Data Type	NET Data Type	Size in Bytes	Low Value	High Value
sbyte	System.Sbyte	1	-128	127
byte	System.Byte	1	0	255
short	System.Int16	2	-32,768	32,767
ushort	System.UInt16	2	0	65,535
int	System.Int32	4	-2,147,483,648	2,147,483,647
uint	System.UInt32	4	0	4,294,967,295
long	System.Int64	8	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807
ulong	System.UInt64	8	0	18,446,744,073 709,551,615
char	System.Char	2	0	65,535
float	System.Single	4	1.5×10^{-45}	3.4×10^{38}
double	System.Double	8	5.0×10^{-324}	$1.7 \times 10^{10^{308}}$
bool	System.Boolean	1	false (0)	true (1)
decimal	System.Decimal	16	1.0×10^{-28}	approx. 7.9×10^{28}

Q&A

Q Why shouldn't all numbers be declared as the larger data types instead of the smaller data types?

A Although it might seem logical to use the larger data types, this would not be efficient. You should not use any more system resources (memory) than you need.

Q What happens if you assign a negative number to an unsigned variable?

A You get an error by the compiler saying you can't assign a negative number to an unsigned variable if you do this with a literal. If you do a calculation that causes an

unsigned variable to go below 0, you get erroneous data. On later days, you will learn how to check for these erroneous values.

Q A decimal value is more precise than a float or a double value. What happens with rounding when you convert from these different data types?

A When converting from a float, double, or decimal to one of the whole number variable types, the value is rounded. If a number is too big to fit into the variable, an error occurs.

When a double is converted to a float that is too big or too small, the value is represented as infinity or 0, respectively.

When a value is converted from a float or double to a decimal, the value is rounded. This rounding occurs after 28 decimal places and occurs only if necessary. If the value being converted is too small to be represented as a decimal, the new value is set to 0. If the value is too large to store in the decimal, an error occurs.

For conversions from decimal to float or double, the value is rounded to the nearest value the float or double can hold. Remember, a decimal has better precision than a float or a double. This precision is lost in the conversion.

3

Q What other languages adhere to the Common Type System (CTS) in the Common Language Runtime (CLR)?

A Microsoft Visual Basic .Net (version 7) and Microsoft Visual C++ .NET (version 7) both support the CTS. Additionally there are versions of a number of other languages that are ported to the CTS. These include Python, COBOL, Perl, Java, and more. Check out the Microsoft Web site for additional languages.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. What are the by value data types available in C#?
2. What is the difference between a signed and unsigned variable?
3. What is the smallest data type you can use to store the number 55?
4. What is the biggest number that a type short variable can hold?
5. What numeric value is the character B?

6. How many bits in a byte?
7. What literal values can be assigned to a Boolean variable?
8. Name three of the reference data types.
9. Which floating-point data type has the best precision?
10. What .NET data type is equivalent to the C# int data type?

Exercises

1. Change the range of values in Listing 3.6 to print the lowercase letters.
2. Write the line of code that declares a variable named xyz of type float and assign the value of 123.456 to it.
3. Which of the following variable names are valid?
 - a) X
 - b) PI
 - c) 12months
 - d) sizeof
 - e) nine
4. **BUG BUSTER:** The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
1: //Bug Buster
2: //-----
3: using System;
4:
5: class variables
6: {
7:     public static void Main()
8:     {
9:         double my_double;
10:        decimal my_decimal;
11:
12:        my_double = 3.14;
13:        my_decimal = 3.14;
14:
15:        Console.WriteLine("\nMy Double: {0}", my_double);
16:        Console.WriteLine("\nMy Decimal: {0}", my_decimal);
17:
18:    }
19: }
```

5. **ON YOUR OWN:** Write a program that declares two variables of each data type and assigns the values 10 and 1.879 to each variable.

WEEK 1

DAY 4

Working with Operators

Now that you know how to store information in variables, you'll want to do something with that data. Most likely you'll want to manipulate the data by making changes to it. For example, you might want to use the radius of a circle to find the area of the circle. Today you

- Discover the types and categories of operators available in C#
- Manipulate information using the different operators
- Change program flow using the `if` command
- Understand which operators have precedence over others
- Explore bitwise operations—if you're brave enough

Types of Operators

Operators are used to manipulate information. You have actually used a number of operators in the programming examples on previous days. Operators are used to perform operations such as addition, multiplication, comparison, and more.

Operators can be broken into a number of categories:

- The Basic assignment operator
- Mathematical/arithmetic operators
- Relational operators
- The Conditional operator
- Other operators (type, size)

Each of these categories and the operators within them are covered in detail in today's lessons. In addition to these categories, it is also important to understand the structure of operator statements. There are three types of operator structures:

- Unary
- Binary
- Ternary

Unary Operator Types

Unary operators are operators that impact a single variable. For example, to have a negative 1, you type

-1

If you have a variable called x, you change the value to a negative by using

-x

The negative requires only one variable, so it is unary. The format of a unary variable will be one of the following depending on the specific operator:

[operator][variable]

or

[variable][operator]

Binary Operator Types

Whereas unary uses only one variable, binary operator types work with two variables. For example, the addition operator is used to add two values together. The format of the binary operator types is

[variable1][operator][variable2]

Examples of binary operations in action are

5 + 4

3 - 2

100.4 - 92348.67

You will find that most of the operators fall into the binary operator type.

Ternary Operator Types

Ternary operators are the most complex operator types to work with. As the name implies, this type of operator works on three variables. C# has only one true ternary operator, the conditional operator. You will learn about it later today. For now, know that ternary operators work with three variables.

Punctuators

Before jumping into the different categories and the specific operators within C#, it is important to understand about punctuators. Punctuators are a special form of operator that helps you format your code, do multiple operations at once, and simply signal information to the compiler. The punctuators that you need to know about are

- **Semicolon** ;. The primary use of the semicolon is to end each C# statement. A semicolon is also used with a couple of the C# statements that control program flow. You will learn about the use of the semicolon with the control statements on Day 5, "Control Statements."
- **Comma** ,. The comma is used to stack multiple commands on the same line. You saw the comma in use on Day 3, "Storing Information with Variables," in a number of the examples. The most common time to use the comma is when declaring multiple variables of the same type:
`int var1, var2, var3;`
- **Parentheses** (). Parentheses are used in multiple places. You will see later in today's lesson, that you can use parentheses to force the order of execution. Additionally, parentheses are used with functions.
- **Braces** {}. Braces are used to group pieces of code. You have seen braces used to encompass classes in many of the examples. You should also have noticed that braces are always used in pairs.

4

Punctuators operate the same way punctuation within a sentence operates. For example, you end a sentence with a period or another form of punctuation. In C#, you end a "line" of code with a semicolon or other punctuator. Line is in quotation marks because a line of code might actually take up multiple lines in a source listing. As you learned on Day 2, "Understanding C# Programs," whitespace and newlines are ignored.



You will find that you can also use braces within the routines you create to block off code. The code put between two braces, along with the braces, is called a *block*.

The Basic Assignment Operator

The first operator that you need to know about is the basic assignment operator, which is an equal sign (=). You've seen this operator already in a number of the examples in earlier lessons.

The basic assignment operator is used to assign values. For example, to assign the value 142 to the variable x, you type

```
x = 142;
```

This compiler takes the value that is on the right side of the assignment operator and places it in the variable on the left side. Consider the following:

```
x = y = 123;
```

This might look a little weird; however, it is legal C# code. The value on the right of the equal sign is evaluated. In this case, the far right is 123, which is placed into the variable y. Then the value of y is placed into the variable x. The end result is that both x and y equal 123.



You cannot do operations on the left side of an assignment operator. For example, you can't do

```
1 + x = y;
```

nor can you put literals or constants on the left side of an assignment operator.

Mathematical/Arithmetic Operators

Now that you are aware of the punctuators, it is time to jump into the operators. Among the most commonly used operators are the mathematical operators. All the basic math functions are available within C#, including addition, subtraction, multiplication, division, and modulus. Additionally, there are compound operators that make doing some of these operations even more concise.

Note

The modulus operator is also known as the *remaindering operator*.

Adding and Subtracting

The additive operators within C# are used for addition and subtraction. For addition, the plus operator (+) is used. For subtraction, the minus (-) operator is used. The general format of using these variables is

```
NewVal = Val1 + Val2;  
NewVal2 = Val1 - Val2;
```

In the first statement, `Val2` is being added to `Val1` and the result is placed in `NewVal`. When this command is done, `Val1` and `Val2` remain unchanged. Any preexisting values in `NewVal` are overwritten with the result.

For the subtraction statement, `Val2` is subtracted from `Val1` and the result is placed in `NewVal2`. Again, `Val1` and `Val2` remain unchanged, and the value in `NewVal2` is overwritten with the result.

`Val1` and `Val2` can be any of the value data types, constants, or a literal. You should note that `NewVal` must be a variable; however, it can be the same variable as `Val1` or `Val2`. For example, the following is legal as long as `Var1` is a variable:

```
Var1 = Var1 - Var2;
```

In this example, the value in `Var2` is subtracted from the value in `Var1`. The result is placed into `Var1`, thus overwriting the previous value that `Var1` held. The following example is also valid:

```
Var1 = Var1 - Var1;
```

In this example, the value of `Var1` is subtracted from the value of `Var1`. Because these values are the same, the result is `0`. This `0` value is then placed into `Var1` overwriting any previous value.

If you want to double a value, you enter the following:

```
Var1 = Var1 + Var1;
```

`Var1` is added to itself and the result is placed back into `Var1`. The end result is that you double the value in `Var1`.

Multiplicative Operators

An easier way to double the value of a variable is to multiply it by two. There are three multiplicative operators commonly used in C#.

Multiplying

The first multiplicative operator is the multiplier (or times) operator, which is an asterisk (*). To multiply two values, you use the following format:

```
NewVal = Val1 * Val2;
```

For example, to double the value in Val1 and place it back into itself (as seen with the last addition example), you can enter the following:

```
Val1 = Val1 * 2;
```

This is the same as

```
Val1 = 2 * Val2;
```

Dividing

To do division, you use the divisor operator, which is a forward slash (/):

```
NewVal = Val1 / Val2;
```

This example divides Val1 by Val2 and places the result in NewVal. To divide 2 by 3, you write the following:

```
answer = 2 / 3;
```

Working with Remainders

There are times when doing division that you want only the remainder. For example, I know that 3 will go into 4 one time; however, I also would like to know that I have 1 remaining. You can get this remainder using the remaindering (also called modulus) operator, which is the percentage sign (%). For example, to get the remainder of 4 divided by 3, you enter:

```
Val = 4 % 3;
```

The result is that Val is 1.

Consider another example that is near and dear to my heart. You have 3 pies that can be cut into 6 pieces, and 13 people each want pie. How many pieces of pie are left over?

To solve this, you first determine how many pieces of pie you have:

```
PiecesOfPie = 3 * 6; // three pies times six pieces
```

The value of `PiecesOfPie` should be obvious to you—18. Now to determine how many pieces are left, you use the modulus operator:

```
PiecesForMe = PiecesOfPie % 13;
```

This sets the value of `PiecesForMe` to $18 \bmod 13$, which is 5. Listing 4.1 verifies this.

LISTING 4.1 pie.cs—Number of Pieces of Pie for Me

```
1: // pie.cs - Using the modulus operators
2: //-----
3: class pie
4: {
5:     static void Main()
6:     {
7:         int PiecesOfPie = 0;
8:         int PiecesForMe = 0;
9:
10:        PiecesOfPie = 3 * 6;
11:
12:        PiecesForMe = PiecesOfPie % 13;
13:
14:        System.Console.WriteLine("Pieces Of Pie = {0}", PiecesOfPie);
15:        System.Console.WriteLine("Pieces For Me = {0}", PiecesForMe);
16:    }
17: }
```

OUTPUT Pieces Of Pie = 18
 Pieces For Me = 5

4

ANALYSIS Listing 4.1 presents the use of the multiplication and modulus operators. Line 10 illustrates the multiplication operator using the same formula as shown earlier.

Line 12 then uses the modulus operator. As you can see from the information printed in lines 14 and 15, the results are as expected.

Compound Arithmetic Assignment Operators

Earlier in today's lesson, you learned about the basic assignment operator. There are also other assignment operators, the compound assignment operators (see Table 4.1).

TABLE 4.1 Compound Arithmetic Assignment Operators

Operator	Description	Non-Compound Equivalent
<code>+=</code>	<code>x += 4</code>	<code>x = x + 4</code>
<code>-=</code>	<code>x -= 4</code>	<code>x = x - 4</code>
<code>*=</code>	<code>x *= 4</code>	<code>x = x * 4</code>

TABLE 4.1 continued

<i>Operator</i>	<i>Description</i>	<i>Non-Compound Equivalent</i>
<code>/=</code>	<code>x /= 4</code>	<code>x = x / 4</code>
<code>%=</code>	<code>x %= 4</code>	<code>x = x % 4</code>

The compound operators provide a concise method for performing a math operation and assigning it to a value. If you want to increase a value by 5, you use the following:

`x = x + 5;`

or you can use the compound operator:

`x += 5;`

As you can see, the compound operator is much more concise.



Tip

Although the compound operators are more concise, they are not always the easiest to understand in code. If you use the compound operators, make sure that what you are doing is clear or remember to comment your code.

Doing Unary Math

All the arithmetic operators you have seen so far have been binary. Each has required two values to operate. There are also a number of unary operators that work with just one value or variable. The unary arithmetic operators are the increment operator `(++)` and the decrement operator `(--)`.

These operators add 1 to the value or subtract 1 from the value of a variable. The following example:

`++x;`

adds 1 to `x`. It is the same as saying

`x = x + 1;`

Additionally, the following:

`--x;`

subtracts 1 from `x`. It is the same as saying

`x = x - 1;`

Listing 4.2 shows the increment and decrement operators in action.

Tip

The increment and decrement operators are handy when you need to step through a lot of values one-by-one.

LISTING 4.2 count.cs—Using the Increment and Decrement Operators

```
1: // count.cs - Using the increment/decrement operators
2: //-----
3:
4: class count
5: {
6:     static void Main()
7:     {
8:         int Val1 = 0;
9:         int Val2 = 0;
10:
11:        System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
12:
13:        ++Val1;
14:        --Val2;
15:
16:        System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
17:
18:        ++Val1;
19:        --Val2;
20:
21:        System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
22:    }
23: }
```

OUTPUT

```
Val1 = 0  Val2 = 0
Val1 = 1  Val2 = -1
Val1 = 2  Val2 = -2
```

ANALYSIS

This listing doesn't do anything spectacular; however, it does illustrate the increment and decrement operators. As you can see in lines 8 and 9, two variables are initialized to `0`. Line 11 prints this so that you can see they are actually `0`. Lines 13 and 14 then increment and decrement each of the variables. Line 16 prints these values so you can see that the actions occurred. Lines 18 to 21 repeat this process.

Using Pre- and Post-Increment Operators

The increment and decrement operators have a unique feature that causes problems for a lot of newer programmers. Assume that the value of `x` is `10`. Look at the following line of code:

```
y = ++x;
```

After this statement executes, what will the values of x and y be? You should be able to guess that the value of x will be 11 after it executes. The value of y will also be 11. Now consider the following line of code. Again consider the value of x to start at 10.

```
y = x++;
```

After this statement executes, what will the values of x and y be? If you said they would both be 11 again, you are wrong! After this line of code executes, x will be 11; however, y will be 10. Confused?

It is simple. The increment operator can operate as a pre-increment operator or a post-increment operator. If it operates as a pre-increment operator, the value is incremented before everything else. If it operates as a post-increment operator, it happens after everything else. How do you know if it is pre or post? Easy. If it is before the variable, `++x`, it is pre. If it is after the variable, `x++`, it is post. The same is true of the decrement operator. Listing 4.3 illustrates the pre and post operations of the increment and decrement operators.

LISTING 4.3 prepost.cs—Using the Increment and Decrement Unary Operators

```
1: // prepost.cs - Using pre- versus post-increment operators
2: //-----
3:
4: class prepost
5: {
6:     static void Main()
7:     {
8:         int Val1 = 0;
9:         int Val2 = 0;
10:
11:        System.Console.WriteLine("Val1 = {0}  Val2 = {1}", Val1, Val2);
12:
13:        System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
14:                               ++Val1, Val2++);
15:
16:        System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
17:                               ++Val1, Val2++);
18:
19:        System.Console.WriteLine("Val1 (Pre) = {0}  Val2 = (Post) {1}",
20:                               ++Val1, Val2++);
21:    }
22: }
```

OUTPUT

```
Val1 = 0  Val2 = 0
Val1 (Pre) = 1  Val2 = (Post) 0
Val1 (Pre) = 2  Val2 = (Post) 1
Val1 (Pre) = 3  Val2 = (Post) 2
```

ANALYSIS It is important to understand what is happening in Listing 4.3. In lines 8 and 9, two variables are again being initialized to 0. These values are printed in line 11.

As you can see from the output, the result is that `Val1` and `Val2` equal 0. Line 13, which continues to line 14, prints the values of these two variables again. The values printed, though, are `++Val1` and `Val2++`. As you can see, the pre-increment operator is being used on `Val1` and the post-increment operator is being used on `Val2`. The results can be seen in the output. `Val1` is incremented by 1 and then printed. `Val2` is printed and then incremented by 1. Lines 16 and 19 repeat these same operations two more times.

Do	DON'T
DO use the compound operators to make your math routines concise.	DON'T confuse the post-increment and pre-increment operators. Remember the pre-increment adds prior to the variable, and the post-increment adds after.

Relational Operators

Questions are a part of life. In addition to asking questions, it is often important to compare things. In programming, you will compare values and then execute code based on the answer. The relational operators are used to compare two values.

With the relational operators, you determine the relationship between two values. The relational operators are listed in Table 4.2.

4

TABLE 4.2 Relational Operators

Operator	Description
<code>></code>	Greater than
<code><</code>	Less than
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

When making comparisons with relational operators, you get one of two results: true or false. Consider the following comparisons made with the relational operators:

`5 < 10` 5 is less than 10, so this is true

`5 > 10` 5 is not greater than 10, so this is false

`5 == 10` 5 does not equal 10, so this is false

`5 != 10` 5 does not equal 10, so this is true

As you can see, each of these results is either true or false. Knowing that you can check the relationship of values should be great for programming. The question is, how do you use these relations?

The `if` Statement

The value of relational operators is that they can be used to make decisions. These decisions are used to change the flow of the execution of your program. The `if` keyword can be used with the relational operators to change the program flow.

The `if` keyword is used to compare two values. The standard format of the `if` command is

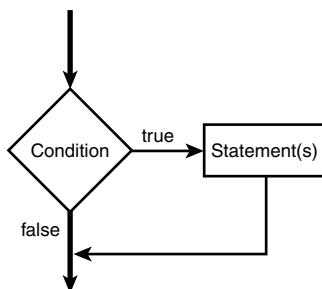
```
if( val1 [operator] val2)
    statement(s);
```

Where *operator* is one of the relational operators; *val1* and *val2* are variables, constants, or literals; and *statement(s)* is a single statement or a block containing multiple statements. Remember a block is one or more statements between brackets.

If the comparison of *val1* to *val2* is true, the statements are executed. If the comparison of *val1* to *val2* is false, the statements are skipped. Figure 4.1 illustrates how the `if` command works.

FIGURE 4.1

The if command.



Applying this to an example helps make this clear. Listing 4.4 presents simple usage of the `if` command.

LISTING 4.4 iftest.cs—Using the `if` Command

```
1: // iftest.cs- The if statement
2: //-----
3:
```

LISTING 4.4 continued

```
4: class iftest
5: {
6:     static void Main()
7:     {
8:         int Val1 = 1;
9:         int Val2 = 0;
10:
11:        System.Console.WriteLine("Getting ready to do the if...");
12:
13:        if (Val1 == Val2)
14:        {
15:            System.Console.WriteLine("If condition was true");
16:        }
17:        System.Console.WriteLine("Done with the if statement");
18:    }
19: }
```

OUTPUT Getting ready to do the if...
Done with the if statement

ANALYSIS This listing uses the `if` statement in line 13 to compare two values to see whether they are equal. If they are, it prints line 15. If not, line 15 is skipped. Because the values assigned to `Val1` and `Val2` in lines 8 and 9 are not equal, the `if` condition fails and line 15 is not printed.

4

Change line 13 to

```
if (Val1 != Val2)
```

Rerun the listing. This time, because `Val1` does not equal `Val2`, the `if` condition evaluates to true. The following is the output:

```
Getting ready to do the if...
If condition was true
Done with the if statement
```



There is not a semicolon at the end of the first line of the `if` command. For example, the following is incorrect:

```
if( x != x);
{
    // Statements to do when the if evaluates to true (which will
    never happen)
}
```

x should always equal x, so x != x will be false and the line // Statements to do when the if evaluates to true... should never execute. Because there is a semicolon at the end of the first line, the if statement is ended. This means that the next statement after the if statement will be executed—the line //Statements to do when the if evaluates to true.... This line will always execute regardless of whether the if evaluates to true or, as in this case, to false. Make sure you don't make the mistake of including a semicolon at the end of the first line of an if statement.

Conditional Logical Operators

The world is rarely a simple place. In many cases you will want to do more than one comparison to determine whether a block of code should be executed. For example, you might want to execute some code if a person is a female and at least 21 years old. To do this, you execute an if statement within another if statement. The following pseudocode illustrates this:

```
if( sex == female )
{
    if( age >= 21)
    {
        // The person is a female that is 21 years old or older.
    }
}
```

There is an easier way to accomplish this—by using a conditional logical operator.

The conditional logical operators enable you to do multiple comparisons with relational operators. The two conditional logical operators that you will use are the AND operator (&&) and the OR operator (||).

The Conditional AND Operator

There are times when you want to verify that a number of conditions are all met. The previous example was one such case. The logical AND operator (&&) enables you to verify that all conditions are met. You can rewrite the previous example as follows:

```
If( sex == female && age >= 21)
{
    // This person is a female that is 21 years old or older.
}
```

You can actually place more than two relationships within a single if statement. Regardless of the number of comparisons, the comparisons on each side of the AND (&&) must be true. For example:

```
if( x < 5 && y < 10 && z > 10)
{
    // statements
}
```

The statements line is reached only if all three conditions are met. If any of the three conditions in the if statements are false, the statements are skipped.

The Conditional OR Operator

There are also times when you do not want all the conditions to be true: instead, you need only one of a number of conditions to be true. For example, you might want to execute some code if the day of week is Saturday or Sunday. In these cases, you use the logical OR operator (||). The following illustrates this with pseudocode:

```
if( day equals sunday OR day equals saturday )
{
    // do statements
}
```

In this example, the statements are executed if the day equals either sunday or saturday. Only one of these conditions needs to be true for the statements to be executed. Listing 4.5 presents both the logical AND and OR in action.

4

LISTING 4.5 and.cs—Using the Logical AND and OR

```
1: // and.cs- Using the conditional AND and OR
2: //-----
3:
4: class andclass
5: {
6:     static void Main()
7:     {
8:         int day = 1;
9:         char sex = 'f';
10:
11:        System.Console.WriteLine("Starting tests... (day:{0}, sex:{1})",
12:                               day, sex );
13:
14:        if ( day >= 1 && day <=7 )           //day from 1 to 7?
15:        {
16:            System.Console.WriteLine("Day is from 1 to 7");
17:        }
18:        if (sex == 'm' || sex == 'f' )   // Male or female?
19:        {
20:            System.Console.WriteLine("Sex is male or female.");
21:        }
22:
23:        System.Console.WriteLine("Done with the checks.");
```

LISTING 4.5 continued

```
24:      }
25: }
```

OUTPUT

```
Starting tests... (day:1, sex:f)
Day is from 1 to 7
Sex is male or female.
Done with the checks.
```

ANALYSIS

This listing illustrates both the `&&` and `||` operators. In line 14 you can see the AND operator (`&&`) in action. For this `if` statement to evaluate to `true`, the day must be greater than or equal to 1 as well as less than or equal to 7. If the day is 1, 2, 3, 4, 5, 6, or 7, the `if` condition evaluates to `true` and line 16 prints. Any other number results in the `if` statement evaluating to `false`, and line 16 will be skipped.

Line 18 shows the OR (`||`) operator in action. Here, if the value in `sex` is equal to the character '`m`' or the character '`f`', line 20 is printed; otherwise, line 20 is skipped.

**Caution**

Be careful with the `if` condition in line 18. This checks for the characters '`m`' and '`f`'. Notice these are lowercase values, which are not the same as the uppercase values. If you set `sex` equal to '`F`' or '`M`' in line 9, the `if` statement in line 18 would still fail.

Change the values in lines 8 and 9 and rerun the listing. You'll see that you get different output results based on the values you select. For example, change lines 8 and 9 to the following:

```
8:      int  day = 9;
9:      char sex = 'x';
```

Here are the results of rerunning the program:

OUTPUT

```
Starting tests... (day:9, sex:x)
Done with the checks.
```

There are also times when you will want to use the AND (`&&`) and the OR (`||`) commands together. For example, you might want to execute code if a person is 21 and is either a male or female. This can be accomplished by using the AND and OR statements together. You must be careful when doing this, though. An AND operator expects the values on both sides of it to be true. An OR statement expects one or the other value to be true. For the previous example, you might be tempted to enter the following:

```
if( age >= 21 AND gender == male OR gender == FEMALE)
    // statement
```

This will not accomplish what you want. If the person is 21 or older and a female, the statement will not execute. The AND portion will result in being `false`. To overcome this problem, you can force how the statements are evaluated using the parenthesis punctuator. To accomplish the desired results, you would change the previous example to

```
if( age >= 21 AND (gender == male OR gender == female))
    // statement
```

The execution always starts with the innermost parenthesis. In this case, the statement `(gender == male OR gender == female)` is evaluated first. Because this uses OR, this portion of the statement will evaluate to `true` if either side is true. If this is true, the AND will compare the age value to see whether the age is greater than or equal to 21. If this proves to be true as well, the statement will execute.

 **Tip**

Use parentheses to make sure that you get code to execute in the order you want.

Do	DON'T
DO use parentheses to make complex math and relational operations easier to understand.	DON'T confuse the assignment operator (<code>=</code>) with the relational equals operator (<code>==</code>).

4

Logical Bitwise Operators

There are three other logical operators that you might want to use: the logical bitwise operators. Although the use of bitwise operations is beyond the scope of this book, I've included a section near the end of today's lesson called "For Those Brave Enough." This section explains bitwise operations, the three logical bitwise operators, and the bitwise shift operators.

The bitwise operators obtain their name from the fact that they operate on bits. A bit is a single storage location that stores either an on or off value (equated to 0 or 1). In the section at the end of today's lesson, you learn how the bitwise operators can be used to manipulate these bits.

Type Operators

As you begin working with classes and interfaces later in this book, you will have need for the type operators. Without understanding interfaces and classes, it is hard to fully

understand these operators. For now, be aware that there are a number of operators that you will need later. These type operators are

- `typeof`
- `is`
- `as`

The `sizeof` Operator

You have already seen the `sizeof` operator. This `sizeof` operator is used to determine the size of a value. You saw the `sizeof` operator in action on Day 3.



Because the `sizeof` operator manipulates memory directly, avoid its use if possible.

The Conditional Operator

There is one ternary operator in C#, the conditional operator. The conditional operator has the following format:

```
Condition ? if_true_statement : if_false_statement;
```

As you can see, there are three parts to this operation, with two symbols used to separate them. The first part of the command is a condition. This is just like the conditions you created earlier for the `if` statement. This can be any condition that results in either `true` or `false`.

After the condition is a question mark. The question mark separates the condition from the first of two statements. The first of the two statements executes if the condition is `true`. The second statement is separated from the first with a colon and is executed if the condition is `false`. Listing 4.6 presents the conditional operator in action.

The conditional operator is used to create concise code. If you have a simple `if` statement that evaluates to doing a simple `true` and `false` statement, the conditional operator can be used. Otherwise, you should avoid the use of the conditional operator. Because it is just a shortcut version of an `if` statement, you should just stick with using the `if` statement itself. Most people reviewing your code will find the `if` statement easier to read and understand.

LISTING 4.6 cond.cs—The Conditional Operator in Action

```
1: // cond.cs - The conditional operator
2: //-----
3:
4: class cond
5: {
6:     static void Main()
7:     {
8:         int Val1 = 1;
9:         int Val2 = 0;
10:        int result;
11:
12:        result = (Val1 == Val2) ? 1 : 0;
13:
14:        System.Console.WriteLine("The result is {0}", result);
15:    }
16: }
```

OUTPUT The result is 0

ANALYSIS This listing is very simple. In line 12, the conditional operator is executed and the result is placed in the variable, `result`. Line 14 then prints this value. In this case, the conditional operator is checking to see whether the value in `Val1` is equal to the value in `Val2`. Because 1 is not equal to 0, the false result of the conditional is set. Modify line 8 so that `Val2` is set equal to 1 and then rerun this listing. You will see that because 1 is equal to 1, the result will be 1 instead of 0.

4

Caution

The conditional operator provides a shortcut for implementing an `if` statement. Although it is more concise, it is not always the easiest for to understand. When using the conditional operator, you should verify that you are not making your code harder to understand.

Understanding Operator Precedence

You've learned about a lot of different operators in today's lessons. Rarely are these operators used one at a time. Often, multiple operators will be used in a single statement. When this happens, a lot of issues seem to arise. Consider the following:

```
Answer = 4 * 5 + 6 / 2 - 1;
```

What is the value of `Answer`? If you said 12, you are wrong. If you said 44 you are also wrong. The answer is 22.

NEW TERM There is a set order in which different types of operators are executed. This set order is called *operator precedence*. The word *precedence* is used because some operators have a higher level of precedence than others. In the example, multiplication and division have a higher level of precedence than addition and subtraction. This means that 4 is multiplied by 5 and 6 is divided by 2, before any addition occurs.

Table 4.3 lists all the operators. The operators at each level of the table are at the same level of precedence. In almost all cases, there is no impact on the results. For example, $5 * 4 / 10$ is the same whether 5 is multiplied by 4 first or 4 is divided by 10.

TABLE 4.3 Operator Precedence

Level	Operator Types	Operators
1	Primary operators	() . [] x++ x-- new typeof sizeof checked unchecked
2	Unary	+ - ! ~ ++x -x
3	Multiplicative	* / %
4	Additive	+ -
5	Shift	<< >>
6	Relational	< > <= >= is
7	Equality	== !=
8	Logical AND	&
9	Logical XOR	^
10	Logical OR	
11	Conditional AND	&&
12	Conditional OR	
13	Conditional	?:
14	Assignment	= *= /= %= += -= <=>=&=^= =

Changing Precedence Order

You learned how to change the order of precedence by using parentheses punctuators earlier in today's lessons. Because parentheses have a higher level of precedence than the operators, what is enclosed in them is evaluated before operators outside of them. Using the earlier example, you can force the addition and subtraction to occur first by using parentheses:

```
Answer = 4 * ( 5 + 6 ) / ( 2 - 1 );
```

Now what will Answer be? Because the parentheses are evaluated first, the compiler first resolves the code to

```
Answer = 4 * 11 / 1;
```

The final result is 44. You can also have parentheses within parentheses. For example, the code could be written as

```
Answer = 4 * ( ( 5 + 6 ) / ( 2 - 1 ) );
```

The compiler would resolve this as

```
Answer = 4 * ( 11 / 1 );
```

Then it would resolve it as

```
Answer = 4 * 11;
```

and finally as the Answer of 44. In this case, the parentheses didn't cause a difference in the final answer; however, there are times when they do.

Converting Data Types

4

When you move a value from one variable type to another, a conversion must occur. Additionally, if you want to perform an operation on two different data types, a conversion might also need to occur. There are two types of conversions that can occur: implicit and explicit.

NEW TERM *Implicit* conversions happen automatically without error. You've read about many of these within today's lesson. What happens when an implicit conversion is not available? For example, what if you want to put the value stored in a variable of type `long` into a variable of type `int`?

NEW TERM *Explicit* conversions are conversions of data that are forced. For the value data types that you learned about today, the easiest way to do an explicit conversion is with a cast. A *cast* is the forcing of a data value to another data type. The format of a cast is

```
ToVariable = (datatype) FromVariable;
```

where `datatype` is the data type you want the `FromVariable` converted to. Using the example of converting a `long` variable to an `int`, you enter the following statement:

```
int IntVariable = 0;  
long LongVariable = 1234;  
IntVariable = (int) LongVariable;
```

In doing casts, you take responsibility for making sure that the variable can hold the value being converted. If the receiving variable cannot store the received value, truncation or other changes can occur. There are a number of times when you are going to need to do explicit conversions. Table 4.4 contains a list of those times.



Explicit conversions as a group also encompass all the implicit conversions. It is possible to use a cast even if an implicit conversion is available.

Table 4.4. Required Explicit Conversions

From Type	To Type(s)
sbyte	byte, ushort, uint, ulong, or char
byte	sbyte or char
short	sbyte, byte, ushort, uint, ulong, or char
ushort	sbyte, byte, short, or char
int	sbyte, byte, short, ushort, uint, ulong, or char
uint	sbyte, byte, short, ushort, int, or char
long	sbyte, byte, short, ushort, int, uint, ulong, or char
ulong	sbyte, byte, short, ushort, int, uint, long, or char
char	sbyte, byte, short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal
decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double

Understanding Operator Promotion

NEW TERM Implicit conversions are also associated with *operator promotion*, which is the automatic conversion of an operator from one type to another. When you do basic arithmetic operations on two variables, they are converted to the same type before doing the math. For example, if you add a byte variable to an int variable, the byte variable is promoted to an integer before being added.

A numeric variable smaller than an `int` will be promoted to an `int`. The order of promotion after an `int` is

```
int  
uint  
long  
ulong  
float  
double  
decimal
```

For Those Brave Enough

For those brave enough, the following sections explain using the bitwise operators. This includes using the shift operators and the logical bitwise operators. Before understanding how these operators work, you need to understand how variables are truly stored.



Tip

It is valuable to understand the bitwise operators and how memory works; however, it is not critical to your understanding C#. This is considered an advanced topic by a lot of people.

4

Storing Variables in Memory

To understand the bitwise operators, you must first understand bits. In yesterday's lesson on data types, you learned that the different data types take different numbers of bits to store. For example, a `char` data type takes 2 bytes. An integer takes 4 bytes. You also learned that there were maximum and minimum values that could be stored in these different data types.

Recall that a byte is 8 bits of memory. Two bytes is 16 bits of memory—two times eight. Four bytes is therefore 32 bits of memory. So the key to all of this is to understand what a bit is.

NEW TERM

A bit is simply a single storage unit of memory that can be either turned on or turned off just like a light bulb. If storing information on a magnetic medium, a bit can be stored as either a positive charge or a negative charge. If working with something such as a CD-ROM, this can be stored as a bump or as an indent. In all these cases, one value is equated to 0 and the other is equated to 1.

If a bit can store only a 0 or a 1, you are obviously very limited in what can be stored. To be able to store larger values, you use bits in groups. For example, if you use 2 bits, you can actually store four numbers, 00, 01, 10, and 11. If you use three bits, you can store 8 numbers, 000, 001, 010, 011, 100, 101, 110, and 111. If you use four bits, you can store 16 numbers. In fact x bits can store 2^x numbers, so a byte (8 bits), can store 2^8 or 256 numbers. Two bytes can store 2^{16} or 65536 values.

Translating from these 1s and 0s is simply a matter of using the binary number system. Appendix D, “Understanding Different Number Systems” explains how you can work with the binary number system in detail. For now understand that the binary system is simply a number system.

You use the decimal number system to count. Where the decimal system uses 10 numbers (0 to 9) the binary system uses two numbers. When counting in the decimal system, you use 1s, 10s, 100s, 1000s, and so forth. For example, the number 13 is one 10 and three 1s. The number 25 is two 10s and five 1s.

The binary system works the same way except there are only two numbers, 0 and 1. Instead of 10s and 100s, you have 1s, 2s, 4s, 8s, and so on. In fact each group is based on taking 2 to the power of a number. The first group is 2 to the power of 0, the second is 2 to the power of 1, the third is 2 to the power of 3, and so on. Figure 4.2 illustrates this.

FIGURE 4.2

Binary versus decimal.

10^3 ... Thousands	10^2 Hundreds	10^1 Tens	10^0 Ones	} Decimal
2^4 ... Sixteens	2^3 Eights	2^2 Fours	2^1 Twos	

2^4 ... Sixteens	} Binary
2^3 Eights	

Presenting numbers in the binary system works the same way it does in the decimal system. The first position on the right is 1s, the second position from the right is 2s, the third is 4s, and so on. Consider the following number:

1101

To convert this binary number to decimal, you can take each value in the number times its positional value. For example, the value in the right column (1s) is 1. The 2s column contains a 0, the 4s column contains a 1, and the 8s column contains a 1. The result is

$$1 + (0 * 2) + (1 * 4) + (1 * 8)$$

The final decimal result is

$$1 + 0 + 4 + 8$$

which is 13. So 1101 in binary is equivalent to 13 in decimal. This same process can be applied to convert any binary number to decimal. As numbers get larger you need more bit positions. To keep things simpler, memory is actually separated into 8-bit units—bytes.

Shift Operators

C# has two shift operators that can be used to manipulate bits. These operators do exactly what their names imply—they shift the bits. The shift operators can shift the bits to the right using the `>>` operator or to the left using the `<<` operator. These operators shift the bits within a variable by a specified number of positions. The format is

```
New_value = Value [shift-operator] number-of-positions;
```

where `Value` is a literal or variable, `shift-operator` is either the right (`>>`) or left (`<<`) shift operator, and `number-of-positions` is how many positions you want to shift the bits. For example, if you have the number 13 stored in a byte, you know its binary representation is

00001101

If you use the shift operator on this, you change the value. Consider the following:

```
00001101 >> 2
```

This shifts the bits in this number to the right two positions. The result is

00000011

This binary value is equivalent to the value of 3. In summary, $13 >> 2$ equals 3. Consider another example:

```
00001101 << 8
```

This example shifts the bit values to the left 8 positions. Because this is a single byte value, the resulting number is 0.

4

Logical Operators

In addition to being able to shift bits, you can also combine the bits of two numbers. There are four bitwise logical operators, as shown in Table 4.5.

TABLE 4.5 Logical Bitwise Operators

Operator	Description
	Logical OR bitwise operator
&	Logical AND bitwise operator
^	Logical XOR bitwise operator
~	Logical NOT bitwise operator

Each of these operators is used to combine the bits of two binary values together. Each has a different result.

The Logical OR Bitwise Operator

When combining two values with the logical OR bitwise operator (`|`), you get the following results:

- If both bits are 0, the result is 0.
- If either or both bits are 1, the result is 1.

Combining two byte values results in the following:

Value 1:	00001111
Value 2:	11001100
<hr/>	
Result:	11001111

The Logical AND Bitwise Operator

When combining two values with the logical AND bitwise operator (`&`), you get the following result:

- If both bits are 1, the result is 1.
- If either bit is 0, the result is 0.

Combining two byte values results in the following:

Value 1:	00001111
Value 2:	11001100
<hr/>	
Result:	00001100

The Logical NOT Operator

When combining two values with the logical NOT bitwise operator (`^`), you get the following result:

- If both bits are the same, the result is 0.
- If one bit is 0 and the other is 1, the result is 1.

Combining two byte values results in the following:

Value 1:	00001111
Value 2:	11001100
<hr/>	
Result:	11000011

Listing 4.7 illustrates the three logical bitwise operators.

LISTING 4.7 The Bitwise Operators

```
1: // bitwise.cs - Using the bitwise operators
2: //-----
3:
4: class bitwise
5: {
6:     static void Main()
7:     {
8:         int ValOne = 1;
9:         int ValZero = 0;
10:        int newVal;
11:
12:        // Bitwise NOT Operator
13:
14:        newVal = ValZero ^ ValZero;
15:        System.Console.WriteLine("\nThe NOT Operator: \n  0 ^ 0 = {0}",
16:                               newVal);
17:
18:        newVal = ValZero ^ ValOne;
19:        System.Console.WriteLine("  0 ^ 1 = {0}", newVal);
20:
21:        newVal = ValOne ^ ValZero;
22:        System.Console.WriteLine("  1 ^ 0 = {0}", newVal);
23:
24:        newVal = ValOne ^ ValOne;
25:        System.Console.WriteLine("  1 ^ 1 = {0}", newVal);
26:
27:        // Bitwise AND Operator
28:
29:        newVal = ValZero & ValZero;
30:        System.Console.WriteLine("\nThe AND Operator: \n  0 & 0 = {0}",
31:                               newVal);
32:
33:        newVal = ValZero & ValOne;
34:        System.Console.WriteLine("  0 & 1 = {0}", newVal);
35:
36:        newVal = ValOne & ValZero;
37:        System.Console.WriteLine("  1 & 0 = {0}", newVal);
38:
39:        newVal = ValOne & ValOne;
40:        System.Console.WriteLine("  1 & 1 = {0}", newVal);
41:
42:        // Bitwise OR Operator
43:
```

LISTING 4.7 continued

```
41:  
42:     newVal = ValZero | ValZero;  
43:     System.Console.WriteLine("\nThe OR Operator: \n  0 | 0 = {0}",  
➥newVal);  
44:  
45:     newVal = ValZero | ValOne;  
46:     System.Console.WriteLine("  0 | 1 = {0}", newVal);  
47:  
48:     newVal = ValOne | ValZero;  
49:     System.Console.WriteLine("  1 | 0 = {0}", newVal);  
50:  
51:     newVal = ValOne | ValOne;  
52:     System.Console.WriteLine("  1 | 1 = {0}", newVal);  
53:   }  
54: }
```

OUTPUT

The NOT Operator:

```
0 ^ 0 = 0  
0 ^ 1 = 1  
1 ^ 0 = 1  
1 ^ 1 = 0
```

The AND Operator:

```
0 & 0 = 0  
0 & 1 = 0  
1 & 0 = 0  
1 & 1 = 1
```

The OR Operator:

```
0 | 0 = 0  
0 | 1 = 1  
1 | 0 = 1  
1 | 1 = 1
```

ANALYSIS

Listing 4.7 is a long listing; however, it summarizes the logical bitwise operators.

Lines 8 and 9 define two variables and assign the values 1 and 0 to them. These two variables are then used repeatedly with the bitwise operators. A bitwise operation is done, and then the result is written to the console. You should review the output and see that the results are exactly as described in the earlier sections.

Summary

Today's lesson presents a lot of information regarding operators and their use. You have learned about the types of operators, including arithmetic, multiplicative, relational, logical, and conditional. You also learned the order in which operators are evaluated

(operator precedence). Finally, today’s lesson ended with an explanation of bitwise operations and the bitwise operators.

Q&A

Q How important is it to understand operators and operator precedence?

A You will use the operators in almost every application you create. Operator precedence is critical to understand. As you saw in today’s lesson, if you don’t understand operator precedence, you might end up with results different from what you expect.

Q Today’s lesson covered the binary number system briefly. Is it important to understand this number system? Also, what other number systems are important?

A Although it is not critical to understand binary, it *is* important. With computers today, information is stored in a binary format. Whether it is a positive versus negative charge, a bump versus an impression, or some other representation, all data is ultimately stored in binary. Knowing how the binary system works will make it easier for you to understand these actual storage values.

In addition to binary, many computer programmers also work with octal and hexadecimal. Octal is a base 8 number system and hexadecimal is a base 16 number system. Appendix D, “Understanding Number Systems,” covers this in more detail.

4

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you’ve learned. Try to understand the quiz and exercise answers before continuing to the next day’s lesson. Answers are provided in Appendix A, “Answers.”

Quiz

The following quiz questions will help verify your understanding of today’s lessons.

1. What character is used for multiplication?
2. What is the result of $10 \% 3$?
3. What is the result of $10 + 3 * 2$?
4. What are the conditional operators?
5. What C# keyword can be used to change the flow of a program?

6. What is the difference between a unary operator and a binary operator?
7. What is the difference between an explicit data type conversion versus an implicit conversion?
8. Is it possible to convert from a long to an integer?
9. What are the possible results of a conditional operation?
10. What do the shift operators do?

Exercises

Please note that answers will not be provided for all exercises. The exercises will help you apply what you have learned in today's lessons.

1. What is the result of the following operation?
 $2 + 6 * 3 + 5 - 2 * 4$
2. What is the result of the following operation?
 $4 * (8 - 3 * 2) * (0 + 1) / 2$
3. Write a program that checks to see whether a variable is greater than 65. If the value is greater than 65, print the statement "The value is greater than 65!".
4. Write a program that checks to see whether a character contains the value of 't' or 'T'.
5. Write the line of code to convert a long called `MyLong` to a short called `MyShort`.
6. **BUG BUSTER:** The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages? What is the error?

```
1: class exercise
2: {
3:     static void Main()
4:     {
5:         int value = 1;
6:
7:         if ( value > 100 );
8:         {
9:             System.Console.WriteLine("Number is greater than 100");
10:        }
11:    }
12: }
```

7. Write the line of code to convert an integer, `IntVal`, to a short, `ShortVal`.
8. Write the line of code to convert a decimal, `DecVal`, to a long, `LongVal`.
9. Write the line of code to convert an integer, `ch`, to a character, `charVal`.

WEEK 1

DAY 5

Control Statements

You've learned a lot in the previous four days. This includes knowing how to store information, knowing how to do operations, and even knowing how to avoid executing certain commands by using the `if` statement. You have learned a little about controlling the flow of a program using the `if` statement; however, there are often times where you need to be able to control the flow of a program even more. Today you

- See the other commands to use for program flow control
- Explore how to do even more with the `if` command
- Learn to switch between multiple options
- Investigate how to repeat a block of statements multiple times
- Discover how to abruptly stop the repeating of code

Controlling Program Flow

By controlling the flow of a program, you are able to create functionality that results in something useful. As you continue to program, you will want to

change the flow in a number of additional ways. You will want to repeat a piece of code a number of times, skip a piece of code altogether, or switch between a number of different pieces of code. Regardless of how you want to change the flow of a program, C# has an option for doing it. Most of the changes of flow can be categorized into two types:

- Selection statements
- Iterative statements

Using Selection Statements

Selection statements enable you to execute specific blocks of code based on the results of a condition. The `if` statement that you learned about previously is a selection statement, as is the `switch` statement.

Revisiting `if`

You've learned about the `if` statement; however, it is worth revisiting. Consider the following example:

```
if( gender == 'm' || gender == 'f' )
{
    System.Console.WriteLine("The gender is valid");
}
if( gender != 'm' && gender != 'f' )
{
    System.Console.WriteLine("The gender value, {0} is not valid", gender);
}
```

This example uses a character variable called `gender`. The first `if` statement checks to see whether `gender` is equal to an '`m`' or an '`f`'. This uses the OR operator you learned about in yesterday's lesson. A second `if` statement prints an error message in the case where the `gender` is not equal to '`m`' or '`f`'. This second `if` statement is an example of making sure that the variable has a valid value. If there is a value other than '`m`' and '`f`', an error message is displayed.

If you look at these two statements and think that something is just not quite optimal, you are correct. C#, like many other languages, offers another keyword that can be used with the `if` statement: the `else` statement. The `else` statement is used specifically with the `if` statement. The format of the `if...else` statement is

```
if ( condition )
{
    // If condition is true, do these lines
}
else
{
```

```
// If condition is false, do these lines
}
// code after if... statement
```

The `else` statement gives you the ability to have code that will be executed when the `if` statement's condition fails. You should also note that either the block of code after the `if` or the block of code after the `else` will execute—but not both. After either of these blocks of code is done executing, the program jumps to the first line after the `if...else` condition.

Listing 5.1 presents the gender code from earlier. This time, the code has been modified to use the `if...else` command. As you can see in the listing, this version is much more efficient and easier to follow than the one presented earlier.

LISTING 5.1 ifelse.cs—Using the `if...else` Command

```
1: // ifelse.cs - Using the if...else statement
2: //-----
3:
4: class ifelse
5: {
6:     static void Main()
7:     {
8:         char gender = 'x';
9:
10:        if( gender == 'm' || gender == 'f' )
11:        {
12:            System.Console.WriteLine("The gender is valid");
13:        }
14:        else
15:        {
16:            System.Console.WriteLine("The gender value, {0}, is not valid",
→gender);
17:        }
18:        System.Console.WriteLine("The if statement is now over!");
19:    }
}
```

5

OUTPUT The gender value, x, is not valid
The if statement is now over!

ANALYSIS This listing declares a simple variable called `gender` of type `char` in line 8. This variable is set to a value of '`x`' when it is declared. The `if` statement starts in line 10. Line 10 checks to see whether `gender` is either '`m`' or '`f`'. If it is, a message is printed in line 12 saying that `gender` is valid. If `gender` is not '`m`' or '`f`', the `if` condition fails and control is passed to the `else` statement in line 14. In this case, `gender` is

equal to 'x', so the `else` command is executed. A message is printed stating that the gender value is invalid. Control is then passed to the first line after the `if...else` statement—line 18.

Modify line 8 to set the value of gender to either '`m`' or '`f`'. Recompile and rerun the program. This time the output will be

OUTPUT The gender is valid
 The if statement is now over!



What would you expect to happen if you set the value of gender to a capital M or F? Remember, C# is case sensitive. A capital letter is not the same thing as a lowercase letter.

Nesting and Stacking `if` Statements

NEW TERM *Nesting* is simply the inclusion of one statement within another. Almost all C# flow commands can be nested within each other.

To nest an `if` statement, you place a second `if` statement within the first. You can nest within the `if` section or the `else` section. Using the gender example, you could do the following to make the statement a little more effective (the nested statement appears in bold):

```
if( gender == 'm' )
{
    // it is a male
}
else
{
    if ( gender == 'f' )
    {
        // it is a female
    }
    else
    {
        //neither a male or a female
    }
}
```

A complete `if...else` statement is nested within the `else` section of the original `if` statement. This code operates just as you expect. If `gender` is not equal to '`m`', the flow goes to the first `else` statement. Within this `else` statement is another `if` statement that starts from its beginning. This second `if` statement checks to see whether the `gender` is

equal to 'f'. If not, the flow goes to the `else` statement of the nested `if`. At that point, you know that gender is neither 'm' nor 'f', and you can add appropriate coding logic.

While nesting makes some functionality easier, you can also stack `if` statements. In the example of checking gender, stacking is actually a much better solution.

Stacking `if` Statements

Stacking `if` statements combines the `else` with another `if`. The easiest way to understand stacking is to see the gender example one more time, stacked (see Listing 5.2).

LISTING 5.2 stacking.cs—Stacking an `if` Statement

```
1: //  stacked.cs - Using the if...else statement
2: //-----
3:
4: class stacked
5: {
6:     static void Main()
7:     {
8:         char gender = 'x';
9:
10:        if( gender == 'm' )
11:        {
12:            System.Console.WriteLine("The gender is male");
13:        }
14:        else if ( gender == 'f' )
15:        {
16:            System.Console.WriteLine("The gender is female");
17:        }
18:        else
19:        {
20:            System.Console.WriteLine("The gender value, {0}, is not valid",
21:                                     gender);
22:        }
23:    }
24: }
```

5

OUTPUT The gender value, x, is not valid
The if statement is now over!

ANALYSIS The code presented in this example is very close to the code presented in the previous example. The primary difference is in line 14. The `else` statement is immediately followed by an `if`. There are no braces or a block. The format for stacking is

```
if ( condition 1 )
{
```

```
// do something about condition 1
}
else if ( condition 2 )
{
    // do something about condition 2
}
else if ( condition 3 )
{
    // do something about condition 3
}
else if ( condition x )
{
    // do something about condition x
else
{
    // All previous conditions failed
}
```

This is relatively easy to follow. With the gender example, you had only two conditions. There are times when you might have more than two. For example, you could create a computer program that checks the roll of a die. You could then do something different depending on what the roll is. Each stacked condition could check for a different number (from 1 to 6) with the final `else` statement presenting an error because there can be only six numbers. The code for this would be

```
if (roll == 1)
    // roll is 1
else if (roll == 2)
    // roll is 2
else if (roll == 3)
    // roll is 3
else if (roll == 4)
    // roll is 4
else if (roll == 5)
    // roll is 5
else if (roll == 6)
    // roll is 6
else
    // it isn't a number from 1 to 6
```

This code is relatively easy to follow because it's easy to see that each of the 6 possible numbers is checked against the roll. If the roll is not one of the 6, the final `else` statement can take care of any error logic or reset logic.



Note

As you can see in the die code, there are no braces used around the `if` statements. If you are using only a single statement within the `if` or the `else`, you don't need the braces. You include them only when you have more than one statement.

The switch Statement

C# provides a much easier way to modify program flow based on multiple values stored in a variable: the `switch` statement. The format of the `switch` statement is

```
switch ( value )
{
    case result_1 :
        // do stuff for result_1
        break;
    case result_2 :
        // do stuff for result_2
        break;
    ...
    case result_n :
        // do stuff for result_x
        break;
    default:
        // do stuff for default case
        break;
}
```

You can see by the format of the `switch` statement that there is no condition. Rather a `value` is used. This value can be the result of an expression, or it can be a variable. This value is then compared to each of the values in each of the `case` statements until a match is found. If a match is not found, the flow goes to the `default` case. If there is not a `default` case, the flow goes to the first statement following the `switch` statement.

When a match is found, the code within the matching `case` statement is executed. When the flow reaches another `case` statement, the `switch` statement is ended. Only one `case` statement will be executed at most. Flow then continues with the first command following the `switch` statement. Listing 5.3 shows the `switch` statement in action, using the earlier example of a roll of a six-sided die.

5

LISTING 5.3 roll.cs—Using the `switch` Statement with the Roll of a Die

```
1: // roll.cs- Using the switch statement.
2: //-----
3:
4: class roll
5: {
6:     public static void Main()
7:     {
8:         int roll = 0;
9:
10:        // The next two lines set the roll to a random number from 1 to 6
11:        System.Random rnd = new System.Random();
12:        roll = (int) rnd.Next(1,7);
```

LISTING 5.3 continued

```
13:  
14:    System.Console.WriteLine("Starting the switch... ");  
15:  
16:    switch (roll)  
17:    {  
18:        case 1:  
19:            System.Console.WriteLine("Roll is 1");  
20:            break;  
21:        case 2:  
22:            System.Console.WriteLine("Roll is 2");  
23:            break;  
24:        case 3:  
25:            System.Console.WriteLine("Roll is 3");  
26:            break;  
27:        case 4:  
28:            System.Console.WriteLine("Roll is 4");  
29:            break;  
30:        case 5:  
31:            System.Console.WriteLine("Roll is 5");  
32:            break;  
33:        case 6:  
34:            System.Console.WriteLine("Roll is 6");  
35:            break;  
36:        default:  
37:            System.Console.WriteLine("Roll is not 1 through 6");  
38:            break;  
39:    }  
40:    System.Console.WriteLine("The switch statement is now over!");  
41: }  
42: }
```

OUTPUT

```
Starting the switch...  
Roll is 1  
The switch statement is now over!
```

Note

Your answer for the roll in the output might be a number other than 1.

ANALYSIS

This listing is a little longer than a lot of the previous listings; however, it is also more functional. The first thing to focus on is lines 16 to 39. These lines contain the `switch` statement that is the center of this discussion. The `switch` statement uses the value stored in the `roll`. Depending on the value, one of the cases will be selected. If the number is something other than 1 though 6, the `default` statement starting in line 39 is

executed. If any of the numbers are rolled (1 through 6), the appropriate `case` statement is executed.

You should note that at the end of each section of code for each `case` statement there is a `break` command, which is required at the end of each set of code. This signals the end of the statements within a `case`. If you don't include the `break` command, you will get a compiler error.

To make this listing a more interesting, lines 11 and 12 were added. Line 11 might look unfamiliar. This line creates a variable called `rnd`, which is an object that will hold a random number. In tomorrow's lesson, you revisit this line of code and learn the details of what it is doing. For now, simply know that it is setting up a variable for a random number.

Line 12 is also a line that will become more familiar over the next few days. The command, `(int) rnd.Next(1,7)` provides a random number from 1 to 6.



Tip

You can use lines 11 and 12 to generate random numbers for any range by simply changing the values from 1 and 7 to the range you want numbers between. The first number is the lowest number that will be returned. The second number is one higher than the highest number that will be returned. For example, if you wanted a random number from 90 to 100, you could change line 12 to:

```
Roll = (int) rnd.Next(90, 101);
```

Multiple Cases for Single Solutions

Sometimes you might want to execute the same piece of code for multiple values. For example, if you want to switch based on the roll of a six-sided die, but you want to do something based only on odd or even numbers, you could group multiple `case` statements. The `switch` statement is

```
switch (roll)
{
    case 1:
    case 3:
    case 5:
        System.Console.WriteLine("Roll is odd");
        break;
    case 2:
    case 4:
    case 6:
        System.Console.WriteLine("Roll is even");
        break;
}
```

```
    default:  
        System.Console.WriteLine("Roll is not 1 through 6");  
        break;  
    }
```

The same code is executed if the roll is 1, 3, or 5. Additionally, the same code is executed if the roll is 2, 4, or 6.



Caution

If you are a C++ programmer, you learned that you could have code execute from multiple case statements by leaving out the break command. This causes the code to drop through to the next case statement. In C#, this is not valid. Code cannot drop through from one case to another. This means that if you are going to group case statements, you cannot place any code between them. You can place it only after the last case statement in each group.

Executing More than One `case` Statement

There are times when you might want to execute more than one case statement within a switch statement. To do this in C#, you can use the goto command. The goto command can be used within the switch statement to go to either a case statement or to the default command. The following code snippet shows the switch statement from the previous section executed with goto statements instead of simply dropping through:

```
switch (roll)  
{  
    case 1:  
        goto case 5;  
        break;  
    case 2:  
        goto case 6;  
        break;  
    case 3:  
        goto case 5;  
        break;  
    case 4:  
        goto case 6;  
        break;  
    case 5:  
        System.Console.WriteLine("Roll is odd");  
        break;  
    case 6:  
        System.Console.WriteLine("Roll is even");  
        break;  
    default:}
```

```
        System.Console.WriteLine("Roll is not 1 through 6");
        break;
    }
```

Although this example illustrates using the `goto`, it is much easier to use the previous example of grouping multiple case statements. You will find there are times, however, when the `goto` provides the solution you need.

Governing Types for `switch` Statements

A `switch` statement has only certain types that can be used. The data type—or the “governing type” for a `switch` statement—is the type that the `switch` statement’s expression resolves to. If this governing type is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or a text string, this type is the governing type. There is another type called an `enum` that is also valid as a governing type. You will learn about `enum` types on Day 8, “Advanced Data Storage: Structures, Enumerators, and Arrays.”

If the data type of the expression is something other than these types, the type must have a single implicit conversion that converts it to a type of `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, or a string. If there isn’t a conversion available, or if there is more than one, you get an error when you compile your program.



Note

If you don’t remember what implicit conversions are, you should review Day 3, “Storing Information with Variables.”

5

Do	DON'T
DO use a <code>switch</code> statement when you are checking for multiple different values in the same variable.	DON'T accidentally put a semicolon after the condition of a <code>switch</code> or <code>if</code> statement: <code>if (condition);</code>

Using Iteration Statements

In addition to changing the flow through selection statements, there are times when you might want to repeat a piece of code multiple times. For times when you want to repeat code, C# provides a number of iteration statements. Iteration statements can execute a block of code zero or more times. Each execution of the code is a single iteration.

The iteration statements in C# are

- while
- do
- for
- foreach

The **while** Statement

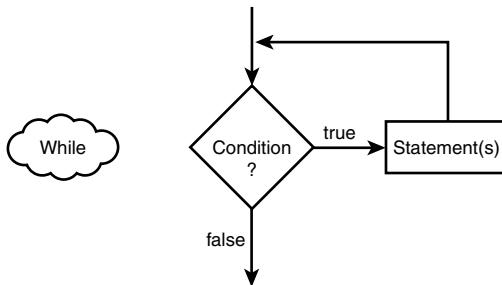
The **while** command is used to repeat a block of code as long as a condition is true. The format of the **while** statement is

```
while ( condition )
    Statement(s)
```

This format is also presented in Figure 5.1.

FIGURE 5.1

The while command.



As you can see from the figure, a **while** statement uses a conditional statement. If this conditional statement evaluates to true, the statement(s) are executed. If the condition evaluates to false, the statements are not executed and program flow goes to the next command following the **while** statement. Listing 5.4 presents a **while** statement that enables you to print the average of 10 random numbers from 1 to 10.

LISTING 5.4 average.cs—Using the **while** Command

```
1: // average.cs Using the while statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //-----
4:
5: class average
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0; // variable to store the running total
```

LISTING 5.4 continued

```
10:     int nbr = 0; // variable for individual numbers
11:     int ctr = 0; // counter
12:
13:     System.Random rnd = new System.Random(); // random number
14:
15:     while ( ctr < 10 )
16:     {
17:         //Get random number
18:         nbr = (int) rnd.Next(1,11);
19:
20:         System.Console.WriteLine("Number {0} is {1}", (ctr + 1), nbr);
21:
22:         ttl += nbr;           //add nbr to total
23:         ctr++;              //increment counter
24:     }
25:
26:     System.Console.WriteLine("\nThe total of the {0} numbers is {1}",
27:     ↪ctr, ttl);
27:     System.Console.WriteLine("\nThe average of the numbers is {0}",
28:     ↪ttl/ctr );
28: }
29: }
```

Note

The numbers in your output will differ from those shown here. Because random numbers are assigned, each time you run the program, the numbers will be different.

5

OUTPUT

```
Number 1 is 2
Number 2 is 5
Number 3 is 4
Number 4 is 1
Number 5 is 1
Number 6 is 5
Number 7 is 2
Number 8 is 5
Number 9 is 10
Number 10 is 2
```

The total of the 10 numbers is 37

The average of the numbers is 3

ANALYSIS

This listing uses the code for random numbers that you saw earlier in today's lesson. Instead of a random number from 1 to 6, this code picks numbers from

1 to 10. You see this in line 18, where the value of 10 is multiplied against the next random number. Line 13 initialized the random variable before it was used in this manner.

The `while` statement starts in line 15. The condition for this `while` statement is a simple check to see whether a counter is less than 10. Because the counter was initialized to `0` in line 11, this condition evaluates to true, so the statements within the `while` will be executed. This `while` statement simply gets a random number from 1 to 10 in line 18 and adds it to the total counter, `tt1` in line 22. Line 23 then increments the counter variable, `ctr`. After this increment, the end of the `while` is reached in line 24. The flow of the program is automatically put back to the `while` condition in line 15. This condition is reevaluated to see whether it is still true. If true, the statements are executed again. This continues to happen until the `while` condition fails. For this program, the failure occurs when `ctr` becomes `10`. At that point, the flow goes to line 25 which immediately follows the `while` statement.

The code after the `while` statement prints the total and the average of the 10 random numbers that were found. The program then ends.



Caution

For a `while` statement to eventually end, you must make sure that you change something in the statement(s) that will impact the condition. If your condition can never be false, your `while` statement could end up in an infinite loop. There is one alternative to creating a false condition: the `break` statement. This is covered in the next section.

Breaking Out Of or Continuing a `while` Statement

It is possible to end a `while` statement before the condition is set to false. It is also possible to end an iteration of a `while` statement before getting to the end of the statements.

To break out of a `while` and thus end it early, you use the `break` command. A `break` immediately takes control to the first command after the `while`.

You can also cause a `while` statement to jump immediately to the next iteration. This is done by using the `continue` statement. The `continue` statement causes the program's flow to go to the condition statement of the `while`. Listing 5.5 illustrates both the `continue` and `break` statements within a `while`.

LISTING 5.5 even.cs—Using `break` and `continue`

```
1: // even.cs- Using the while with the break and continue commands.  
2: //-----  
3:
```

LISTING 5.5 continued

```
4: class even
5: {
6:     public static void Main()
7:     {
8:         int ctr = 0;
9:
10:        while (true)
11:        {
12:            ctr++;
13:
14:            if (ctr > 10 )
15:            {
16:                break;
17:            }
18:            else if ( (ctr % 2) == 1 )
19:            {
20:                continue;
21:            }
22:            else
23:            {
24:                System.Console.WriteLine("...{0}...", ctr);
25:            }
26:        }
27:        System.Console.WriteLine("Done!");
28:    }
29: }
```

OUTPUT

```
...2...
...4...
...6...
...8...
...10...
Done!
```

5**ANALYSIS**

This listing prints even numbers and skips odd numbers. When the value of the counter is greater than 10, the while statement is ended with a break statement.

This listing declares and sets a counter variable, `ctr`, to 0 in line 8. A while statement is then started in line 10. Because a break is used to end the loop, the condition in line 10 is simply set to true. This, in effect, creates an infinite loop. Because this is an infinite loop, a break statement is needed to end the while statement's iterations. The first thing done in the while statement is that `ctr` is incremented in line 12. Line 14 then checks to see whether the `ctr` is greater than 10. If `ctr` is greater than 10, line 16 executes a break statement, which ends the while and sends the program flow to line 27.

If `ctr` is less than 10, the `else` statement in line 18 is executed. This `else` statement is stacked with an `if` statement that checks to see whether the current number is odd. This is done using the modulus operator. If the counter is even, by using the modulus operator with 2, you get a result of 0. If it is odd, you get a result of 1. When an odd number is found, the `continue` statement is called in line 20. This sends control back to the top of the `while` statement where the condition is checked again. Because the condition is always true (literally), the `while` statement's statements are executed again. This starts with the increment of the counter in line 12 again, followed by the checks.

If the number is not odd, the `else` statement in line 22 will execute. This final `else` statement contains a single call to `WriteLine`, which prints the counter's value.

The do Statement

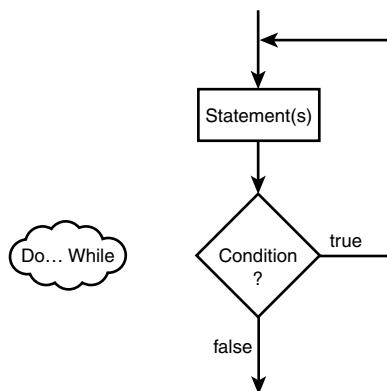
If a `while` statement's condition is false on the initial check, the `while` statements will never execute. There are times, however, when you want statements to execute at least once. For these times, the `do` statement might be a better solution.

The format of the `do` statement is

```
do  
Statement(s)  
while ( condition );
```

This format is also presented in Figure 5.2.

FIGURE 5.2
The do command.



As you can see from the figure, a `do` statement first executes its statements. After executing the statements, a `while` statement is presented with a condition. This `while` statement and condition operate the same as the `while` you explored earlier in listing 5.4. If the condition evaluates to true, program flow returns to the statements. If the condition

evaluates to false, the flow goes to the next line after the do...while. Listing 5.6 presents a do command in action.

Note

Because of the use of the while with the do statement, a do statement is often referred to as a do...while statement.

LISTING 5.6 do_it.cs—The do Command in Action

```
1: // do_it.cs Using the do statement.
2: // Get random numbers (from 1 to 10) until a 5 is reached.
3: //-----
4:
5: class do_it
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0;    // variable to store the running total
10:        int nbr = 0;   // variable for individual numbers
11:        int ctr = 0;   // counter
12:
13:        System.Random rnd = new System.Random(); // random number
14:
15:        do
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            ctr++;           //number of numbers counted
21:            ttl += nbr;      //add nbr to total of numbers
22:
23:            System.Console.WriteLine("Number {0} is {1}", ctr, nbr);
24:
25:        } while ( nbr != 5 );
26:
27:        System.Console.WriteLine("\n{0} numbers were read", ctr);
28:        System.Console.WriteLine("The total of the numbers is {0}", ttl);
29:        System.Console.WriteLine("The average of the numbers is {0}", ttl/ctr
  );
30:    }
31: }
```

5

OUTPUT

```
Number 1 is 1
Number 2 is 6
Number 3 is 5
```

```
3 numbers were read
```

```
The total of the numbers is 12
The average of the numbers is 4
```

ANALYSIS As with the previous listings that used random numbers, your output will most likely be different from what is displayed. You will have a list of numbers, ending with 5.

For this program, you want to do something at least once—get a random number. You want to then keep doing this until you have a condition met—you get a 5. This is a great scenario for the `do` statement. This listing is very similar to an earlier listing. In lines 9 to 11, you set up a number of variables to keep track of totals and counts. In line 13, you again set up a variable to get random numbers.

Line 15 is the start of your `do` statement. The body of the `do` (lines 16 to 24) is executed. First, the next random number is obtained. Again, this is a number from 1 to 10 that is assigned to the variable `nbr`. Line 20 keeps track of how many numbers have been obtained by adding 1 to `ctr` each time a number is read. Line 21 then adds the value of the number read to the total. Remember, the following code:

```
ttl += nbr
```

is the same as

```
ttl = ttl + nbr
```

Line 23 prints the obtained number to the screen with the count of which number it is.

Line 25 is the conditional portion of the `do` statement. In this case, the condition is that `nbr` is not equal to 5. As long as the number obtained, `nbr`, is not equal to 5, the body of the `do` statement will continue to execute. When a 5 is received, the loop ends. If you look in the output of your program, you will find that there is always only one 5, and that it is always the last number.

Lines 27, 28, and 29 prints statistical information regarding the numbers you found.

The `for` Statement

Although the `do...while` and the `while` statement give you all the functionality you really need to control iterations of code, they are not the only commands available. Before looking at the `for` statement, check out the code in the following snippet:

```
ctr = 1;
while ( ctr < 10 )
{
    //do some stuff
    ctr++;
}
```

In looking at this snippet of code, you can see that a counter is used to loop through a `while` statement. The flow if this code is this:

- Step 1: Set a counter to the value of 1.
- Step 2: Check to see whether the counter is less than 10.
 - If the counter is not less than 10 (the condition fails), go to the end.
- Step 3: Do some stuff.
- Step 4: Add 1 to the counter.
- Step 5: Go to Step 2.

These steps are a very common use of iteration. Because this is a common use, you are provided with the `for` statement, which consolidates the steps into a much simpler format:

```
for ( initializer; condition; incrementor )
Statement(s);
```

You should review the format presented here for the `for` statement, which contains three parts within parentheses: the `initializer`, the `condition`, and the `incrementor`. Each of these three parts is separated by a semicolon. If one of these expressions is to be left out, you still need to include the semicolon separators.

The `initializer` is executed when the `for` statement begins. It is executed only once at the beginning and then never again.

After executing the `initializer`, the `condition` statement is evaluated. Just as the condition in the `while` statement, this must evaluate to either true or false. If this evaluates to true, the statement(s) are executed.

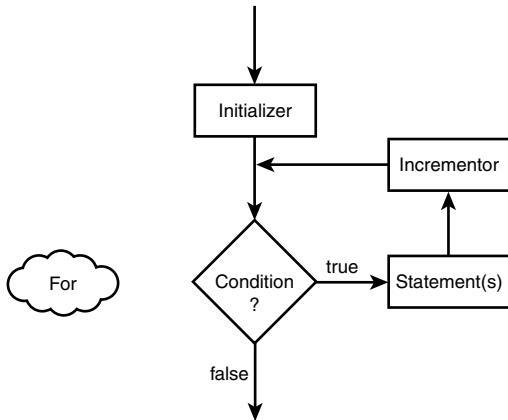
After the statement or statement block executes, program flow is returned to the `for` statement where the `incrementor` is evaluated. This `incrementor` can actually be any valid C# expression; however, it is generally used to increment a counter.

After the `incrementor` is executed, the `condition` is again evaluated. As long as the `condition` remains true, the statements will be executed, followed by the `incrementor`. This continues until the `condition` evaluates to false. Figure 5.3 illustrates the flow of the `for` statement.

Before jumping into a listing, let's revisit the `while` statement that was presented at the beginning of this section:

```
for ( ctr = 1; ctr < 10; ctr++ )
{
    //do some stuff
}
```

FIGURE 5.3
The for statement.



This `for` statement is much simpler than the code used earlier with the `while` statement. The steps that this `for` statement executes are

Step 1: Set a counter to the value of 1.

Step 2: Check to see whether the counter is less than 10.

If the counter is not less than 10 (condition fails), go to the end of the `for` statement.

Step 3: Do some stuff.

Step 4: Add 1 to the counter.

Step 5: Go to Step 2.

These are the same steps that were followed with the `while` statement snippet earlier.

The difference is that the `for` statement is much more concise and easier to follow.

Listing 5.7 presents a more robust use of the `for` statement. In fact, this is the same program you saw in sample code earlier, only now it is much more concise.

LISTING 5.7 foravg.cs—Using the for Statement

```
1: // foravg.cs Using the for statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //-----
4:
5: class average
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0; // variable to store the running total
10:        int nbr = 0; // variable for individual numbers
```

LISTING 5.7 continued

```
11:     int ctr = 0; // counter
12:
13:     System.Random rnd = new System.Random(); // random number
14:
15:     for ( ctr = 1; ctr <= 10; ctr++ )
16:     {
17:         //Get random number
18:         nbr = (int) rnd.Next(1,11);
19:
20:         System.Console.WriteLine("Number {0} is {1}", (ctr), nbr);
21:
22:         ttl += nbr; //add nbr to total
23:     }
24:
25:     System.Console.WriteLine("\nThe total of the 10 numbers is {0}",
26:     =>ttl);
27:     System.Console.WriteLine("\nThe average of the numbers is {0}",
28:     =>ttl/10 );
27: }
28: }
```

OUTPUT

```
Number 1 is 10
Number 2 is 3
Number 3 is 6
Number 4 is 5
Number 5 is 7
Number 6 is 8
Number 7 is 7
Number 8 is 1
Number 9 is 4
Number 10 is 3
```

```
The total of the 10 numbers is 54
```

```
The average of the numbers is 5
```

ANALYSIS

Much of this listing is identical to what you saw earlier in today's lessons. You should note, however, the difference. In line 15, you see the use of the `for` statement. The counter is initialized to 1, which makes it easier to display the value in the `WriteLine` routine in line 20. The condition statement in the `for` statement is adjusted appropriately as well.

What happens when the program flow reaches the `for` statement? Simply put, the counter is set to 1. It is then verified against the condition. In this case, the counter is less than or equal to 10, so the body of the `for` statement is executed. When the body in lines 16 to 23 is done executing, control goes back to the incrementor of the `for` statement in line 15. In this `for` statement's incrementor, the counter is incremented by 1. The condition is then checked again and if true, the body of the `for` statement executes again. This

continues until the condition fails. For this program, this happens when the counter is set to 11.

The `for` Statement Expressions

You can do a lot with the `initializer`, `condition`, and `incrementor`. You can actually put any expressions within these areas. You can even put in more than one expression.

If you use more than one expression within one of the segments of the `for` statement, you need to separate them. The separator control is used to do this. The separator control is the comma. As an example, the following `for` statement initializes two variables and increments both:

```
for ( x = 1, y = 1; x + y < 100; x++, y++ )  
    // Do something...
```

In addition to being able to do multiple expressions, you also are not restricted to using each of the parts of a `for` statement as described. The following example actually does all of the work in the `for` statement's control structure. The body of the `for` statement is an empty statement—a semicolon:

```
for ( x = 0; ++x <= 10; System.Console.WriteLine("{0}", x) )  
    ;
```

This simple line of code actually does quite a lot. If you enter this into a program, it prints the numbers 1 to 10. You're asked to turn this into a complete listing in one of today's exercises at the end of the lesson.



Caution

You should be careful about how much you do within the `for` statement's control structures. You want to make sure you don't make your code too complicated to follow.

The `foreach` Statement

The `foreach` statement iterates in a way similar to the `for` statement. The `foreach` statement, however, has a special purpose. It can loop through collections such as arrays. The `foreach` statement, collections, and arrays are covered on Day 8.

Revisiting `break` and `continue`

The use of `break` and `continue` were presented earlier with the `while` statement. Additionally, you saw the use of the `break` command with the `switch` statement. These two commands can also be used with the other program flow statements.

In the `do...while` statement, `break` and `continue` operate exactly like the `while` statement. The `continue` command loops to the conditional statement. The `break` command sends the program flow to the statement following the `do...while`.

With the `for` statement, the `continue` statement sends control to the incrementor statement. The condition is then checked, and if true, the `for` statement continues to loop. The `break` statement sends the program flow to the statement following the `for` statement.

The `break` command exits the current routine. The `continue` command starts the next iteration.

Using `goto`

The `goto` statement is fraught with controversy, regardless of the programming language you use. Because the `goto` statement can unconditionally change program flow, it is very powerful. With power comes responsibility. Many developers avoid the `goto` statement because it is easy to create code that is hard to follow.

There are three ways the `goto` statement can be used. As you saw earlier, the `switch` statement is home to two of the uses of `goto`: `goto case` and `goto default`. You saw these in action earlier in the discussion on the `switch` statement.

The third `goto` statement takes the following format:

```
goto label;
```

With this form of the `goto` statement, you are sending the control of the program to a label statement.

5

Labeled Statements

A label statement is simply a command that marks a location. The format of a label is `label_name:`

Notice that this is followed by a colon, not a semicolon. Listing 5.8 presents the `goto` statement being used with labels.

LISTING 5.8 Score.cs—Using the `goto` Statement with a Label

```
1: // score.cs    Using the goto and label statements.  
2: // Disclaimer: This program shows the use of goto and label  
3: //                This is not a good use; however, it illustrates  
4: //                the functionality of these keywords.
```

LISTING 5.8 continued

```
5: // -----
6:
7: class score
8: {
9:     public static void Main()
10:    {
11:        int score = 0;
12:        int ctr = 0;
13:
14:        System.Random rnd = new System.Random();
15:
16:        Start:
17:
18:        ctr++;
19:
20:        if (ctr > 10)
21:            goto EndThis;
22:        else
23:            score = (int) rnd.Next(60, 101);
24:
25:        System.Console.WriteLine("{0} - You received a score of {1}",
26:                               ctr, score);
27:
28:        goto Start;
29:
30:    EndThis:
31:
32:    System.Console.WriteLine("Done with scores!");
33: }
34: }
```

OUTPUT

```
1 - You received a score of 83
2 - You received a score of 99
3 - You received a score of 72
4 - You received a score of 67
5 - You received a score of 80
6 - You received a score of 98
7 - You received a score of 64
8 - You received a score of 91
9 - You received a score of 79
10 - You received a score of 76
Done with scores!
```

ANALYSIS

The purpose of this listing is relatively simple; it prints 10 scores that are obtained by getting 10 random numbers from 60 to 100. This use of random numbers is similar to what you've seen before except for one small change. In line 23, instead of starting at 1 for the number to be obtained, you start at 60. Additionally,

because the numbers you want are from 60 to 100, the upper limit is set to 101. By using 101 as the second number, you get a number less than 101.

The focus of this listing is lines 16, 21, 28, and 30. In line 16 you see a label called Start. Because this is a label, the program flow skips past this line and goes to line 18 where a counter is incremented. In line 20, the condition within an if statement is checked. If the counter is greater than 10, a goto statement in line 21 is executed, which sends program flow to the EndThis label in line 30. Because the counter is not greater than 10, program flow goes to the else statement in line 22. The else statement gets the random score in line 23 that was already covered. Line 25 prints the score obtained. Program flow then hits line 28, which sends the flow unconditionally to the Start label. Because the Start label is in line 16, program flow goes back to line 16.

This listing does a similar iteration to what can be done with the while, do, or for statements. In many cases, you will find there are programming alternatives to using goto. If there is a different option, use it first.



Tip

Avoid using the goto whenever possible. It can lead to what is referred to as *spaghetti code*. Spaghetti code is code that winds all over the place and is therefore hard to follow from one end to the next.

Nesting Flow

All of the program flow commands from today can be nested. When nesting program flow commands, make sure that the commands are ended appropriately. You can create a logic error and sometimes a syntax error if you don't nest properly.

5

Do	DON'T
DO comment your code to make it clearer what the program and program flow is doing.	DON'T use a goto statement unless it is absolutely necessary.

Summary

You learned a lot in today's lesson, and you'll use this knowledge in virtually every C# application you create.

In today's lesson, you once again covered some of the constructs that are part of the basic C# language. You first expanded on your knowledge of the if statement by learning about

the `else` statement. You then learned about another selection statement, the `switch`. Selection statements were followed by a discussion of iterative program flow control statements. This included use of the `while`, `do`, and `for` statements. You learned that there is another command called the `foreach` that you will learn about on Day 8. In addition to learning how to use these commands, you also learned that they can be nested within each other. Finally, you learned about the `goto` statement and how it can be used with `case`, `default`, or labels.

Q&A

Q Are there other types of control statements?

A Yes—`throw`, `try`, `catch`, and `finally`. You will learn about these in future lessons.

Q Can you use a text string with a `switch` statement?

A Yes. A string is a “governing type” for `switch` statements. This means that you can use a variable that holds a string in the `switch` and then use string values in the `case` statements. Remember, a string is simply text in quotation marks. In one of the exercises, you create a `switch` statement that works with strings.

Q Why is the `goto` considered so bad?

A The `goto` statement has gotten a bad rap. If used cautiously and in a structured, organized manner, the `goto` statement can help solve a number of programming problems. The use of `goto case` and `goto default` are prime examples of good uses of `goto`. `goto` has a bad rap because the `goto` statement is often not used cleanly; programmers use it to get from one piece of code to another quickly and in an unstructured manner. In an object-oriented programming language, the more structure you can keep in your programs, the better—and more maintainable—they will be.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you’ve learned. Try to understand the quiz and exercise answers before continuing to the next day’s lesson. Answers are provided in Appendix A, “Answers.”

Quiz

1. What commands are provided by C# for repeating lines of code multiple times?
2. What is the fewest times the statements in a `while` will execute?

3. What is the fewest times the statements in a `do` will execute?
4. Consider the following `for` statement:
`for (x = 1; x == 1; x++)`
What is the conditional statement?
5. In the `for` statement in question 4, what is the incrementor statement?
6. What statement is used to end a `case` expression in a `select` statement?
7. What punctuation character is used with a label?
8. What punctuation is used to separate multiple expressions in a `for` statement?
9. What is nesting?
10. What command is used to jump to the next iteration of a loop?

Exercises

1. Write an `if` statement that checks to see whether a variable called `file-type` is '`s`', '`m`', or '`j`'. Print the following message based on the `file-type`:
`s The filer is single`
`m The filer is married filing at the single rate`
`j The filer is married filing at the joint rate`
2. Is the following `if` statement valid? If so, what is the value of `x` after this code executes?
`int x = 2;`
`int y = 3;`
`if (x==2) if (y>3) x=5; else x=9;`
3. Write a `while` loop that counts from 99 to 1.
4. Rewrite the `while` loop in exercise 3 as a `for` loop.
5. **BUG BUSTER:** Is the following listing correct? If so, what does it do? If not, what is wrong with the listing (Ex5-5.cs)?

5

```
// Ex5-5.cs. Exercise 5 for Day 5
// -----
class score
{
    public static void Main()
    {
        int score = 99;

        if ( score == 100 );
        {
            System.Console.WriteLine("You got a perfect score!");
        }
    }
}
```

```
    }
    else
        System.Console.WriteLine("Bummer, you were not perfect!");
    }
}
```

6. Create a `for` loop that prints the numbers 1 to 10 all within the initializer, condition, and incrementor sections of the `for`. The body of the `for` should be an empty statement.
7. Write the code for a `switch` statement that switches on the variable `name`. If the name is "Robert", print a message that says "Hi Bob". If the name is "Richard", print a message that says "Hi Rich". If the name is "Barbara", print a message that says "Hi Barb". If the name is "Kalee", print a message that says "You Go Girl!". On any other name, print a message that says "Hi x" where `x` is the person's name.
8. Write a program to roll a six-sided die 100 times. Print the number of times each of the sides of the die was rolled.

CHAPTER 3

A GUIDED TOUR THROUGH C#: PART I

You will learn about the following in this chapter:

- The advantages of applying two important OOP concepts—abstraction and encapsulation
- Why the C# keywords public and private play an important role in implementing encapsulation
- The basic C# elements needed to write simple C# applications
- How to write a user interactive application using simple keyboard input and screen output
- Single line comments and why comments are important in your source code
- The special meaning of keywords
- How to define the beginning and the end of a class and method body by using C#'s block construct
- How to use C#'s if statement to make your program respond in different ways to different user input
- The string class and its ability to let your programs store and process text
- The special role played by the Main method
- The static keyword and why Main must always be declared public and static
- How to use variables
- How to call a method and thereby use its functionality
- Several useful classes from the .NET Framework class libraries and how to reuse these in the C# source code
- Statements in C#—the declaration, assignment, method call, and if statements
- General C# concepts based on the knowledge gained from the C# source code example
- How to access and use the .NET Framework Documentation so you can explore and reuse the .NET Framework's comprehensive collection of classes

Introduction

Each language construct of a C# program does not exist in isolation. It has its own vital part to play but is also closely interrelated with other elements. This makes it difficult to look at any one aspect of C# without requiring the knowledge of others. Due to this circular dependence among the elements of C#, this chapter, along with Chapter 4, “A Guided Tour Through C#: Part II,” and Chapter 5, “Your First Object-Oriented C# Program,” presents an overview of several important features, to give you an introductory feel for the language.

The presentation is facilitated by C# source code examples containing several essential elements of C#. Each element will be presented, discussed, and related to other parts of the C# program in a practically related fashion. This will enable you to start writing your own programs during this chapter. I hope you will grab this opportunity to play with and explore C#. Some of the most important parts of this and the following two chapters are the programming exercises at the end of each chapter. You don’t become a proficient C# programmer just by learning lots of definitions by heart but by doing and unleashing your creativity. So have a go at these exercises and use your imagination to come up with other ideas of how to improve the programs or, even better, create your own programs.

Abstraction and Encapsulation

Object-oriented programming is at the core of C#. Practically speaking, everything in C# is an object. Even the simple program you encountered in Chapter 2, “Your First C# Program” relies on OO principles. So before we dive into the first C# example, it is useful to expand on our OO introduction by looking at two core concepts of OO called abstraction and encapsulation.

Abstraction

Consider an airplane. Initially, it seems an impossible task to represent an airplane in a computer program. Just think about all the details—intricately designed jet engines, extremely complex onboard computers, entertainment systems. It is dizzying. However, by looking at the role we want the airplane to play in our application, we can significantly reduce the features to be represented. Perhaps we just need an airplane to be a position on a map. Perhaps we need to test the aerodynamic characteristics of an airplane in which case we only need to represent its outer shape. Perhaps an interior designer is making a 3D presentation of the airplane’s interiors, in which case, he needs to represent only the interior surfaces of the airplane.

In every airplane role mentioned, it is possible to identify the key characteristics of the airplane relevant to that particular application. Essentially, we are coping with the intricacies of the airplane by abstracting away from them.

Abstraction is one of the fundamental concepts utilized by programmers to cope with complexity.

When an object (or system) is specified or depicted in a simpler, less detailed fashion than its real counterpart, this specification is called an *abstraction*. Highlighting the properties that are relevant to the problem at hand, while ignoring the irrelevant and unduly complicating properties, creates a useful abstraction in OOP.

When creating an abstraction, it is important to include only the features that are part of the object being specified. The behavior of an object must not go beyond that of the expected. For example, creating an abstract car with the ability to draw architectural plans should be avoided. It creates confusion among yourself and other programmers trying to understand your source code.



Note

Recall the elevator simulation discussion from Chapter 2, "Your First C# Program." When designing an elevator simulation program, we must first ignore (by abstraction) all the unnecessary characteristics and parameters of the real world. For example, the color of each elevator and the hairstyle of each person in them can be disregarded. On the other hand, important information is the speed of each elevator, the number of people wanting to catch an elevator, and the floors to which they want to travel.

The simulation enables us to calculate statistics concerning waiting times and eventually an estimate for the number and types of elevators needed to service the building, just as Big Finance Ltd. requested in Chapter 2.

You have already seen one example of an abstraction in the C# program example in Chapter 2. It contained a class called **Shakespeare**.

Lacking arms, legs, inner organs, and a brain, the class had very little resemblance to the writer and person named Shakespeare, but it could still recite one very short quote of Shakespeare. Thus, because a very tiny piece of Shakespeare could be found in it, we still dared to call our class **Shakespeare**. Our **Shakespeare** class was an abstraction of the real writer and person—Shakespeare.

Encapsulation

Whereas abstraction focuses on reducing the complexity of how we view the real world, encapsulation concentrates on the design of our C# source code. It is a powerful mechanism for reducing complexity and protecting data of individual objects on the programming level.

Encapsulation is the process of combining data and the actions (methods) acting on the data into a single entity, just as you saw in Chapter 2. In OOP and hence C#, such an entity is equivalent to an *object*.

Encapsulation is a mechanism for hiding instance variables and irrelevant methods of a class from other objects. Only methods, which are necessary to use an object of the class, are exposed.

The term encapsulation might sound like a sophisticated academic term, but despite this first impression, we can find parallels to it from our everyday life. We might not refer to it as

encapsulation, but the analogies are so striking that an example from the real world might be instructive. The example provided here returns to the world of elevators.

Encapsulation in a Real World Elevator System

Any useful elevator has a mechanism, such as buttons, that allow a passenger to select his or her destination floor. Pressing a button is a very *easy* action to perform for the user of the elevator. However, for the elevator faced with many conflicting simultaneous requests from many different people, it can be a complicated matter deciding which floor to go to and fulfill the request of each traveler expediently. Many conditions complicate the design of an elevator; some examples are provided in the following Note.



The Complicated Life of an Elevator

Modern elevators rely on sophisticated algorithms residing inside computers that control every move they make. Here are a few reasons why it is a complicated matter to decide the next move of an elevator.

A person called A might enter the elevator at floor 3 and request to go to floor 30. However, another person, B, has already pressed the button on floor 1 requesting to be picked up there. If the elevator goes directly to floor 30 with person A (but without B), person A will get there as fast as he or she could ever expect. However, person B has to wait. Perhaps, by merely moving two floors down, the elevator could have taken both people to a higher floor at the same time, saving time and power. Instead, the elevator has to go all the way up to floor 30 and then back down to pick up B. With many people constantly requesting elevators and destination floors, the matter gets much more complex.

At least a couple of other aspects complicate matters further for our trusted elevator:

- Several elevators are probably working together to transport people between the different floors. They consequently need a “team” approach to fulfill as many requests as possible. Densely populated buildings with large amounts of traffic between floors require approximately two elevators for every three floors. Thus, a 60-floor building would require about 40 elevators.
- Certain floors are more likely to be visited and have pick-me-up requests than other floors. These probabilities might vary throughout the day. If the elevators can somehow remember these patterns, they can optimize the efficiency of the sequence of floors visited.

Sophisticated algorithms have been constructed running inside computers that control the movements of the elevators, allowing them to serve as many people as possible, fast and efficiently.

This was a bit of elevator talk, but it leads us to an essential observation.

It is *easy* for a person to *press a button* and thereby use the hidden complex services (algorithms, engines, hydraulics, gears, wires, data about current speed, maximum speed, and so on) provided by the elevator. Thus, the ignorant passenger, in terms of elevator technology, can easily utilize the services of the elevator to accomplish his or her tasks of the day.

Not only is this concealment of elevator data, algorithms, and mechanics an advantage for travelers who do not want to deal with complex matters that are irrelevant to them; but it also improves the reliability of the elevator itself. No unauthorized and incompetent person can

gain access to the inner control mechanisms of the elevator. Imagine if the elevator was equipped with a small “cockpit” where anybody traveling in the elevator could tinker with the maximum speed, or make the elevator believe it was on an incorrect floor, or force it to move to a floor other than the one calculated by its control algorithms. A chaotic situation would soon prevail.

A final advantage of segregating the buttons from the underlying elevator mechanisms is the capability of making changes to these mechanisms without altering the buttons and, hence, the way it is operated. In this way, an elevator can contain the same buttons that everyone has become accustomed to using for many years, while many underlying hardware and software upgrades have taken place to keep the system up to date.

Encapsulation in an Elevator Simulation

Let's look at how the previously discussed issues relate to an object-oriented C# elevator simulation program.

An elevator simulation program written in C# will probably have `Elevator` and `Person` objects along with others. Even though now inside a computer program, a `Person` object will still be able to perform the equivalent of “pressing the button” of an `Elevator` object and giving it a floor request.

Each `Elevator` object of the simulation probably contains, like its real counterpart, many complex methods and many sensitive data. For example, many `Elevator` objects might, to make the simulation particularly realistic, be equipped with algorithms (similar to the algorithms of real world elevators) to calculate the most efficient sequence of floors visited.

Now the important analogy—each `Person` object (and the programmer implementing it) of the simulation should still not need or be allowed to “know” about the inner complexities (source code) of any `Elevator` object. All they should “care” about, just like their real counterparts, is to get from one floor to another. They don't want to be confused about unnecessary complexities either. They should not be allowed to tinker with the inner workings (data and source code) of the `Elevator` object, and the programmers implementing the elevator should be able to change and upgrade the source code without interfering with how the `Elevator` object is used (operated).

To accomplish this concealment in the OO world, we say that we *encapsulate* (hide) the underlying data and source code that is irrelevant for all but the `Elevator` objects themselves.

When we write and create the `Elevator` objects in C#, we somehow need a way to tell the program and the other objects, including the `Person` objects, that the data and source code are hidden. C# has a special word for this purpose called `private`. Thus, by *declaring* the data and relevant source code to be `private`, we are able to prevent any other object from gaining access to these parts of our `Elevator` object.



Convention

Any words with a special meaning in C# as well as references made to parts of source code presented in this book appear in a special font, such as `private`.



private and public

The word **private** has a special meaning to the C# compiler. It is used to hide method and instance variables (residing inside an object) from other objects.

Likewise, the word **public** has a special (albeit opposing) meaning to the C# compiler. **public** is used to allow other objects access to methods and instance variables of a specific object.

Because **private** and **public** control the accessibility of the methods and instance variables belonging to an object, these two words are referred to as *access modifiers*.

However, if we hide *all* the instance variables and methods of an object, none of its methods can be accessed, rendering the object completely useless. For example, a **Person** object needs somehow the ability to “express” its floor request. A method with this capacity could arbitrarily be called **NewFloorRequest**. What would happen if we declared **NewFloorRequest** to be **private**? Well, an **Elevator** object that had just “loaded” a “passenger” (a **Person** object) with these characteristics would not be able to access the **NewFloorRequest** method, which is the only means to find out the floor to which this “passenger” “wanted” to “travel.” This would be the equivalent of a real person without any arms, who is unable to reach out and press a button. In the C# source code, we somehow need a way to put “arms” on our **Person** object and buttons on our **Elevator** object, so contact can be made between the two objects.

We do this by exposing methods like **NewFloorRequest**. Instead of marking this method with the **private** word, we use another special word from C# called **public**.

A **Person** object can now “walk into” an **Elevator** object. When ready, the **Elevator** object can send a message to the **Person** object and, via the **NewFloorRequest** method, receive the desired destination floor of this **Person**. Figure 3.1 illustrates this scenario. Notice the double lined arrow. It represents the message sent to the **Person** object. The single lined arrow going in the opposite direction represents the value (the floor number request) returned to the **Elevator** object from the **Person** object. The double lined arrow represents a method call. Programmers say that the **Elevator** object is *calling* the **NewFloorRequest** method.

The **Elevator** object, like its real counterpart, might need to somehow keep track of, for example, the speed at which it is traveling and its maximum speed; these are the *instance variables* (data) of the object, as mentioned previously. Just like the elevators of the real world, the **Elevator** objects of the C# simulation do not want any interference from other objects to mistakenly change their instance variables. Therefore, the **Elevator** objects declare them to be **private**, as shown in Figure 3.1, making them inaccessible for other objects.

Encapsulation in a Typical Class

Recall the objects displayed in Figure 2.3 of Chapter 2. They are equivalent to the objects shown in Figure 3.1. A generalized object using C#’s access modifiers and indicating the encapsulating **public** layer is displayed in Figure 3.2. Notice how the **public** area of this generic object can be viewed to surround the **private** area like a protective shell. The object still contains the instructions executed by the computer (flow of execution or algorithms); they have now been divided into a **public** part and a **private** part. All the data (or instance variables) are hidden inside in the **private** area.

FIGURE 3.1
Elevator object requesting information from *Person* object.

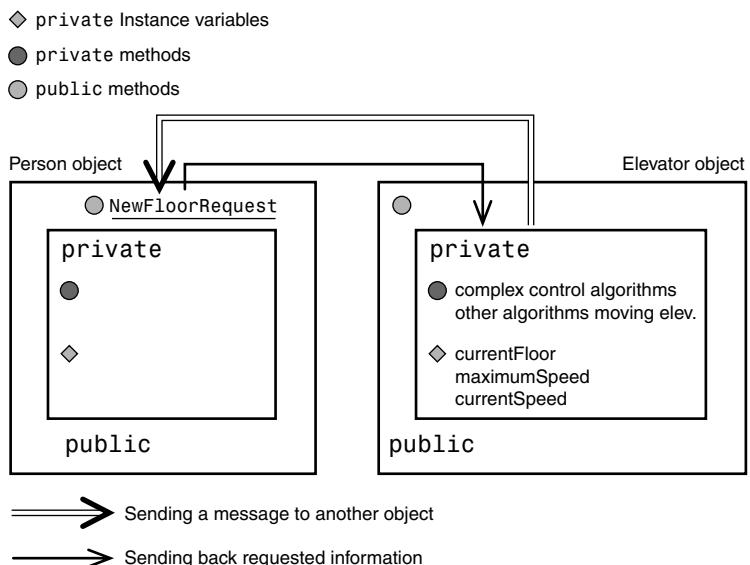
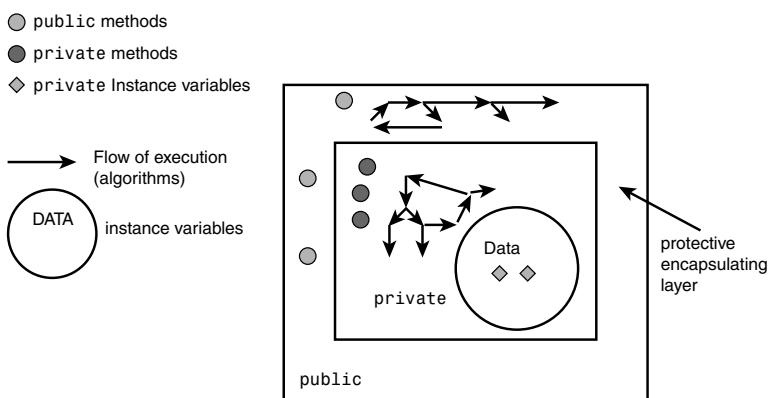


FIGURE 3.2
A generic object.



Class and Object

The class works as a template for the creation of objects. The blueprint of an elevator is to the elevator designer what the class is to the programmer.

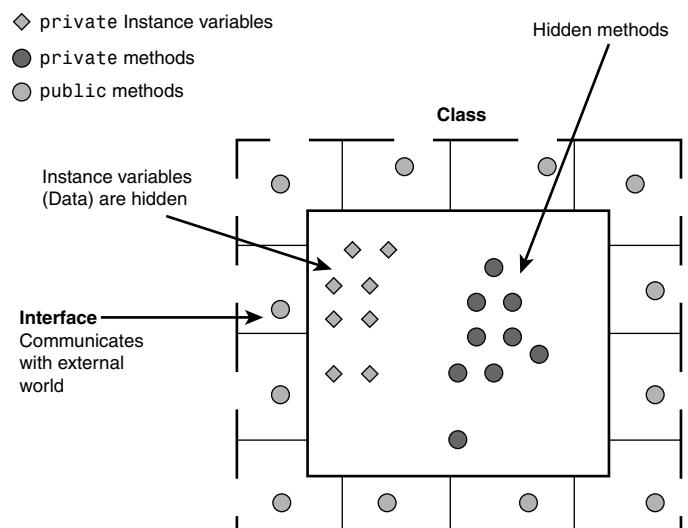
All objects of the same class contain the same methods and the same instance variables. Consequently, a class is a logical construct; but an object has the ability to perform the actions specified by the class.

All objects of the same class are called *instances of a class*. When an object is formed during the execution of a program (as when a tangible elevator is manufactured), we say it is *created* or *instantiated*.

We can now depict a typical generic class as illustrated in Figure 3.3. Simply viewed, the class consists of data (instance variables) and algorithms (methods). When you write the source code for a class, you specify the methods and the instance variables that you want this class, and all objects derived from it, to contain. All instance variables and methods are collectively referred to as *members* or *fields* of the class.

FIGURE 3.3

A generic class illustrates encapsulation.



Encapsulation entails a layer that is meant to communicate with the outside world. Only through this part can the outside world communicate with the object. It further allows for a hidden part to exist inside this layer. This protective layer is denoted an *interface* and should only consist of methods.



Helper Methods

private methods can only be called by methods belonging to that same object. They provide functionality and support for other (often **public**) methods in the object and are frequently called *helper methods* or *utility methods*.



Note

The main advantages of encapsulation are as follows:

- *It provides for an abstraction layer*—Encapsulation saves the programmer who uses a class the need to know about the details of how this particular class is implemented.
- *It decouples the user interface of the object with its implementation*—It is possible to upgrade the implementation of an object while maintaining its user interface.
- *It covers the object with a protective wrapper*—Encapsulation protects the data inside the object from unwanted access that, otherwise, could cause the data to be corrupted and misused.



Only Methods Should Be Part of an Object's Interface, not Instance Variables

Instance variables positioned in the interface (this is possible in C#) effectively break the protective wrapper and allow for other objects to tinker with those exposed pieces of data. If access to specific data is needed, it should always be through a method (or properties and indexers, as discussed in later chapters).

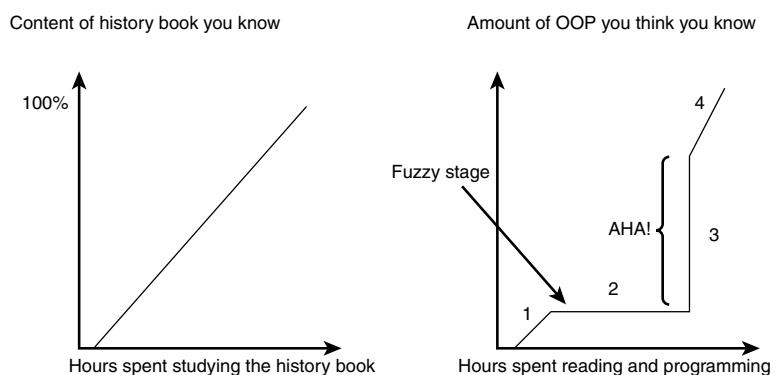
A Note on Learning Object-Oriented Programming

Object-oriented programming and its related concepts might seem a bit fuzzy and overwhelming to you right now. This is perfectly normal, so don't despair.

In my experience, learning OOP is not like learning the content of, say, a history book by heart. While reading the history book, you gradually learn the material. It seems that an hour spent studying gives you an hour's worth of knowledge (unless the subject is complex and interrelated with other historical events), resulting in a nearly linear relationship between the amount of hours spent studying and the amount of knowledge you have acquired (see Figure 3.4).

FIGURE 3.4

Comparing learning by heart and learning OOP.



This is most often not the case when learning OOP because of its interrelated and dynamic aspects and because OOP implies more than just a bunch of statements being executed one after the other. Initially, you will likely learn the names of several concepts (see 1 in Figure 3.4) but without much understanding for their real purpose. You might spend some time in this fuzzy, perhaps frustrating stage. Despite your studying efforts, you might not feel you are really gaining an increased understanding for the underlying logic; new concepts presented seem to confuse rather than enlighten (see 2 in Figure 3.4). However, your mind is constantly absorbing new aspects, information, and subtleties whenever you read the theory and work through the examples. Suddenly, at a specific moment in time (perhaps at night just after waking up from your favorite dream about abstraction and encapsulation), you experience this quantum leap in understanding, this AHA! Experience. All of a sudden, the pieces of the puzzle form a meaningful picture (see 3 in Figure 3.4). Equipped with this fundamental understanding, it becomes much easier to learn the new, more advanced OOP concepts (see 4 in Figure 3.4).

So don't get frustrated—have fun instead. Attack the OOP concepts from different angles, theoretically as well as experimentally, with the C# language.

An Interactive Hello World! Program

While you are digesting the OO information from the previous section, in the next few sections, I will present and analyze a C# source code example accompanied by an introduction to many of the fundamental aspects and features of the C# language. This will be a step toward your first object-oriented C# program presented in the last section of Chapter 5, which constitutes a simple, but working C# elevator simulation source program, directly related to our earlier, somewhat theoretical OO discussion.

Presenting Hello.cs

By tradition, the first program written in a new language prints the two words Hello World! onscreen. Because this is your second program (the first program `Shakespeare.cs` was in Chapter 2), we will write a similar program but slightly more advanced. Instead of merely printing out Hello World!, the user can interact with the program in Listing 3.1 and choose whether Hello World! is printed onscreen.



Note

Any professional interactive application allocates large amounts of source code dealing with invalid user input. Metaphorically speaking, the program attempts to prevent any user from inputting squares where the program expects triangles. To keep the source programs presented in this book compact and focused at the relevant features presented, most parts of the book do not include any user input validation code.

A central facet of programming should be aimed at making programs user friendly. The end user of a program and the programmer are often different people. As the programmer, you cannot expect the user to know how to interact with your program. You must enable the program to guide the user and make the program easy to use, as shown in the sample output after Listing 3.1. There are two possible outputs from the program, depending on whether the user wants Hello World! to be printed. The text typed in by the user is shown in boldface. Notice how the sentence:

Type **y** for yes; **n** for no.

guides the user through the program.

LISTING 3.1 Source Code for Hello.cs

```
01: // This is a simple C# program
02: class Hello
03: {
04:     // The program begins with a call to Main()
05:     public static void Main()
```

LISTING 3.1 continued

```

06:    {
07:        string answer;
08:
09:        System.Console.WriteLine("Do you want me to write the two words?");
10:        System.Console.WriteLine("Type y for yes; n for no. Then <enter>");
11:        answer = System.Console.ReadLine();
12:        if (answer == "y")
13:            System.Console.WriteLine("Hello World!");
14:        System.Console.WriteLine("Bye Bye!");
15:    }
16: }
```

Sample output 1 results when the user answers **y** (yes):

```

Do you want me to write the two words?
Type y for yes; n for no. Then <enter>.
y <enter>
Hello World!
Bye Bye!
```

Sample output 2 results when the user answers **n** (no):

```

Do you want me to write the two words?
Type y for yes; n for no. Then <enter>.
n <enter>
Bye Bye!
```

Listing 3.2 provides a brief analysis of each source code line in Listing 3.1. There is a one-to-one correspondence between the line numbers in the two listings. Each line of Listing 3.1 is thoroughly discussed in the following sections. Listing 3.2 is only meant as a quick reference when you return to Listing 3.1 for a reminder about how a particular construct is written.

LISTING 3.2 Analysis of Source Code for Hello.cs

```

01: Make comment: This is a simple C# program
02: Begin the definition of a class named Hello
03: Begin the block of the Hello class definition
04:     Make comment: The program begins with a call to Main()
05:     Begin the definition of a method called Main()
06:     Begin the block of the Main() method definition
07:         Declare a variable called answer, which can store text
08:         Empty line
09:         Print out: Do you want me to write the two words? Move down
09:             one line
10:         Print out: Type y for yes; n for no. Then <enter> Move down one
10:             line
11:         Store users answer in the answer variable. Move down one line.
12, 13:         If answer stores a 'y' then print: Hello World!
12, 13:         If answer does not store a 'y' then jump over
12, 13:             line 13 and continue with line 14.
14:             Print out: Bye Bye! Move down one line.
15:         End block containing Main() method definition
16:     End block containing Hello class definition
```



Convention

Output printed on the command console from the program is presented as `Bye Bye!`

It's time to start up good old Notepad and type in the source code. You can call the source file `Hello.cs`. Compile `Hello.cs`, run the program by typing `Hello` after the console prompt, and compare the output with what is shown in the sample output from Listing 3.1.



If the Screen Disappears Before You Can Read All the Output

You might have decided to use a text editor other than Notepad, perhaps even an Integrated Development Environment (IDE) configured for C#. This is fine. However, you might encounter a small problem with some systems with the command console disappearing too early, certainly before you ever get a chance to read the onscreen output.

Fortunately, there is a simple solution to this problem. Put the following line of code at the end of the `Main` method (just before the brace that ends the block containing the `Main` method):

```
System.Console.ReadLine();
```

When the program encounters this statement, it will wait for you to press the Enter key, allowing you time to study the output onscreen.

The program in Listing 3.1 is relatively simple, but it contains many essential ingredients of a typical C# program. Let's take a closer look at each part of the program. After you have mastered the concepts presented in this section, they can be applied to most C# programs you will write.

Basic Elements of `Hello.cs`

Every line number contained in this section refers to the line numbers of Listing 3.1.

Comments

Line 1 contains a *comment*. The compiler ignores the contents of a comment. It is used purely to describe or explain the elements of the program to anyone reading through the source code. In this case, the comment simply tells you that the source code you are looking at represents a simple C# program.

```
01: // This is a simple C# program
```



Comments

Comments are extremely valuable for your source code. They are as important as the other elements of the source code. Not only do they enable other people to understand your code, they also act as your own valuable reminders in source code you haven't seen for some time. It is only too easy to forget the structure and logic behind your own source code.

The double forward slash (//) tells the compiler to ignore the rest of the line. In line 1, the comment is on its own line, but it can also be on the same line as code. Consequently lines 1 and 2 could have been combined as follows:

```
class Hello // This is a simple C# program
```

but the following is invalid:

```
// This is a simple C# program class Hello
```

The whole line is suddenly regarded as a comment and so `class Hello` will be ignored completely by the compiler.

Defining a Class

To explain line 2, I must briefly introduce the concept keyword also called reserved word. A *keyword* has a special predefined meaning in the C# language and is recognized by the compiler.

02: class Hello

Line 2 uses the keyword `class` to declare that a class is being defined. `Hello` is the name of the class and must be positioned immediately after `class`.

Every language, including spoken languages, contains words with a special meaning. In English, we call it a vocabulary; in science it's called terminology.

Likewise, the C# language has its own vocabulary made up of *keywords*, also referred to as *reserved words*. Listing 3.1 introduces the following keywords: `class`, `public`, `static`, `void`, `string`, and `if`. It's wise to spend a moment locating these in Listing 3.1. You can find the remaining C# keywords in Appendix C, "Reserved Words in C#," which is found on www.samspublishing.com.

Keywords have very special meanings to the compiler. They cannot, as the term reserved word indicates, be used for other objectives in C#.

`class`, for instance, cannot be used as a name for any element in C#, such as a method or a variable.

Let's give an example from the English language of why this makes sense. Look at the sentence, "Running is very helpful. Running fixed the washing machine today."

Hmm... what is going on here? Well, the parents of Running didn't quite follow the traditional conventions when they named their son Running. Their strange naming preferences distorted our ability to communicate for a moment. Despite our brain's amazing capacity of deciphering information, we still were confused with the use of the word Running. Did Running refer to the English *keyword* running (as a verb) or the son's name Running?

In contrast to our brain, the compiler is a very exact and non-forgiving creature. If for instance line 2 of Listing 3.1 was changed to `class class`; instead of `class Hello`, the compiler would be even more confused than us, causing it to report an error.

Notice that a keyword can be part of a name, so `classVariable` is an acceptable name.

The technical term for a name like `Hello` is *identifier*. Identifiers are, apart from classes, also used to name elements like methods and instance variables.

Had we implemented our `Elevator` class from the elevator simulation, we would most likely have used the identifier `Elevator` instead of `Hello`.



Identifiers (Names)

Names in source code are often called identifiers. Many elements, such as classes, objects, methods and, variables must have identifiers. In contrast to the keywords of C#, which were decided by its designers, you as a programmer decide each identifier.

A few rules apply here. An identifier must consist entirely of letters, digits (0–9), or the underscore character (`_`). An identifier cannot start with a digit and cannot be one of the keywords displayed in Appendix C which you can find on www.samspublishing.com.

Examples:

Legal identifiers:

```
Elevator
_elevator
My2Elevators,
My_Elevator
MyElevator
```

Illegal identifiers:

```
Ele vator
6Elevators
```

C# is case sensitive, so lowercase and uppercase letters are considered to be different characters. `Hello` and `hello` are as different to the compiler as `Hello` and `Bye`.

Braces and Blocks of Source Code

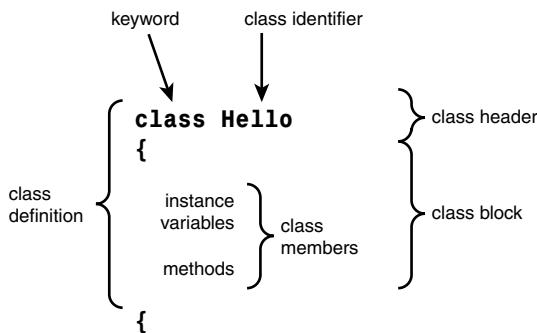
Line 3 contains a brace (`{`). It indicates the beginning of a block.

A *block* is a piece of C# source code enclosed within braces. `}` indicates the end of a block. After a block of code has been created, it becomes a logical unit. Braces, which are used to indicate blocks of code, always work together in pairs. Whenever you see a `{`, you know there must be a matching `}` somewhere. The matching `}` to line 3 is found in line 16. We have another matching pair of braces in lines 6 and 15.

```
03: {
```

Because `{` of line 3 is positioned immediately after line 2, the compiler knows that the entire definition of the `Hello` class is contained in between `{` of line 3 and `}` of line 16. Methods and instance variables can now be inserted in this class definition block, as shown in Figure 3.5, by making sure that all method and instance variable declarations are written between the two braces.

FIGURE 3.5
The class definition.



Tip

The following is a way to help prevent missing braces. Whenever you need to indicate the start of a block with a left brace ({), immediately type the right brace (}) under the left brace. Then position the cursor between the two braces and write the source code for this particular block.

In line 4

```
04:      // The program begins with a call to Main()
```

we recognize the // indicating the beginning of a comment.

The `Main()` Method and Its Definition

Line 5 indicates the beginning of the definition for a method called `Main`. There is no special C# keyword such as “method” indicating that we are dealing with a method. The compiler, though, works this out by recognizing the parentheses () after `Main`.

```
05:      public static void Main()
```

The `Main` method has a special meaning in C#. Any C# application begins its execution at the `Main` method. This method is called by the .NET runtime when the program is started.

For example, a sophisticated spreadsheet program written in C# might contain thousands of methods with different identifiers, but only a method named `Main` will be called by the .NET runtime to start the program.

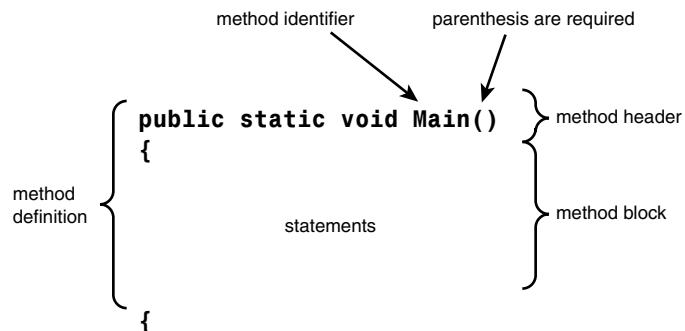
I will not attempt to provide you with the exact meaning of all the parts of line 5, because this involves a detailed understanding of how C# deals with certain object-oriented principles.

However, Figure 3.3, presented earlier in this chapter, can certainly help illuminate parts of line 5. Remember that a class consists of an interface made up of `public` methods and a hidden part consisting of `private` methods and `private` instance variables.

`public`, in line 5, is an access modifier. This keyword allows the programmer to control the visibility of a class member. By using `public` in line 5 somewhere in front of `Main`, we indicate that `Main` is a `public` method and so it is part of the interface of the `Hello` class. As a result, `Main` can be called from outside a `Hello` object.

Figure 3.6 illustrates the major elements of a method definition.

FIGURE 3.6
A method definition.



The `Main` Method

Every C# program must contain a method called `Main`. When a program is executed, the .NET runtime will look for a `Main` method as a first step. When found, it will execute this method; thus, it is always the first method to be executed in a program. An error is reported if no `Main` method is found.

`Main` is in Listing 3.1 located inside of the `Hello` class and the .NET runtime is located outside. When .NET attempts to execute the `Main` method, it will be regarded as just another object trying to access a method of the class. Therefore, we need to expose the `Main` method and make it part of the interface of its class.

A `Main` method must always be declared `public` to let .NET gain access to it.

Typically `Main` will cause other methods located in other objects to be executed but, as you can see, our first simple examples merely contain one class with one `Main` method.

To get an initial feel for the meaning of the `static` keyword, recall our discussion about the differences between a class and an object. A class is just a specification of how to create an object, just like an architectural drawing is only a plan of how to create a real house. A class usually cannot take any actions. Well, the `static` keyword lets us cheat a bit here. It enables us to use methods of a class without instantiating any objects first.

When `static` is part of a method header, we tell the class that it does not need to be instantiated for an object outside the class to use the method. In this case, it allows us to specify that `Main` can be used without first instantiating a specific object based on the `Hello` class. This is necessary here because `Main` is called by the .NET runtime before any objects are created.



Note

A `Main` method must always be declared `public` and `static`.

If the meaning of `static` seems a bit fuzzy and incomprehensible, don't worry. We will return to this concept in more detail later.

To fully understand the meaning of `void` in line 5, you need to know a bit more about how methods work. At this point, I will only give you a brief explanation. `void` indicates that `Main()` does not return a value to the caller.

In line 6, `{` indicates the beginning of `Main()`'s block, which contains the source code comprising the method. The matching `}` ending this block can be found in line 15.

```
06:     {
```



Tip

Choose meaningful variable names in your source code to improve clarity and readability. Avoid abbreviations; don't be afraid to choose long variable names. Which is clearer to you, `avgSpPHr` or `averageSpeedPerHour`?

Source code that can be understood simply by reading it rather than having to consult manuals and look through excessive amounts of comments is said to be *self-documenting*.

Variables: Representing Data Stored in Computer Memory

`answer` in line 7 is a variable. `answer` is the identifier of this variable.

A *variable* is a named memory location representing a stored piece of data. The keyword `string` dictates `answer` to be of type `string`.

```
07:     string answer;
```

The programmer can arbitrarily choose the identifier `answer`, whereas `string` is unchangeable because it is a keyword.

By typing `answer` after `string`, as in line 7, we say in technical terms that we have declared `answer` to be a variable of type `string`.



Note

Any variable used in a C# program must be declared.

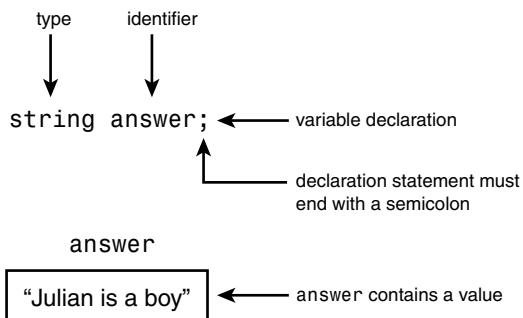
We use the `answer` variable in lines 11 and 12. A variable of type `string` can contain text. For example, "Coco is a dog," "Julian is a boy," "y," or "n" are all examples of text and so are valid data storable in `answer`. In C#, strings of characters are denoted with " " (double quotes). I have illustrated the parts that make up a variable, such as `answer`, in Figure 3.7.

The figure shows that a variable consists of three things:

- Its *identifier*, which is `answer` in this case.

FIGURE 3.7

A variable's type, identifier, and value.



- Its *type* or the kind of information it can hold, which in this case is `string`, meaning a sequence of characters.
- Its *value*, which is the information presently stored. In the illustration, the current value is `"Julian is a boy"`.

I have not revealed all the facts about the `string` type here. String is actually a *reference* type so does not itself hold the text it seems to store; it merely refers to the memory location holding the text. For now, however, we can happily continue disregarding this knowledge. We will return to this aspect in Chapter 6, “Types Part I: The Simple Types.”

The only thing left to explain in line 7 is the semicolon character (`;`). Any task accomplished by a C# program can be broken down into a series of instructions. A simple instruction is called a *statement*. All statements must be terminated with a semicolon character. Line 7 is a declaration statement so a semicolon character must terminate the line.

Line 8 is an empty line.

08:

The C# compiler will simply ignore an empty line. The empty line is inserted to improve the look and readability of the source code.

Invoking Methods of the .NET Framework

The program in line 9

09: `System.Console.WriteLine("Do you want me to write the two words?");`

instructs the computer to print out the following:

`Do you want me to write the two words?`

For now, you can consider `System.Console.WriteLine` to be a peculiar way of saying “print whatever is shown in parenthesis after `WriteLine` onscreen and then move down one line.”

Briefly, this is what happens behind the curtains of line 9. `System.Console` is a class from the .NET Framework. Recall that the .NET Framework is a class library containing numerous useful classes written by professional programmers from Microsoft. Here, we are essentially reusing a class referred to as `System.Console` in our program to print out text onscreen.

`System.Console` contains a method, named `WriteLine`, that is called with the command `System.Console.WriteLine`. `WriteLine` carries out an action; it prints the text inside the parenthesis ("Do you want me to write the two words?").

When a method is called to perform a task, we say the method is being *invoked*. The item inside the parenthesis (in this case, the text: "Do you want me to write the famous words?") is called an *argument*. An *argument* provides information needed by the invoked method to carry out its task. The argument is passed to the `WriteLine` method when invoked. `WriteLine` then has access to this data for its own statements. We can then describe the action of `WriteLine` as follows, "When `WriteLine` is invoked, print out the argument passed to it."

Line 9 is a statement, like line 8, and so it must end with a semicolon.

There is one problem with line 9 that we briefly need to address. It might not be obvious, but we are using a method of the class `System.Console` in line 9. How can we use the method of a class? Several times now we have emphasized classes as mere plans and objects as the doers. However, by using the keyword `static` (mentioned earlier) to cheat a little bit, it becomes possible to use the method of a class. The programmer who implemented the `WriteLine` method used `static` to make `WriteLine` available for use without first having to instantiate `System.Console`.

The General Mechanics of Method Invocation

The instructions of a method reside inside its method definition in the form of statements.

To invoke a method means to initiate the statements it has been instructed to perform. They will be performed in a sequential manner, beginning at the topmost statement and in the same sequence as they are written in the source code.

Methods can only be defined inside classes. A method is an action that an object is able to perform. It is invoked by writing the object name (or the class name if the method is declared `static`) followed by a period (full stop) ., called a *dot*, followed by the method name and ending with a pair of parenthesis () that may or may not contain arguments. Arguments are data passed to the method.

A call to a non-static method residing in an object generally looks like the following:

ObjectName.*MethodName*(*Optional.Arguments*)

A call to a static method includes the class name as in the following:

ClassName.*MethodName*(*Optional.Arguments*)

By substituting the general terms with actual names, we can form the following valid statement, which is identical to line 14 of Listing 3.1.

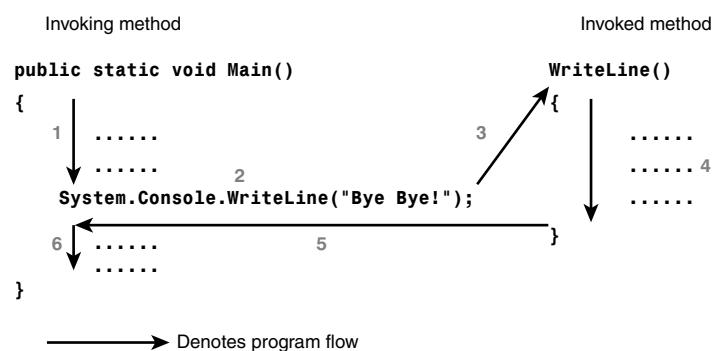
```
System.Console.WriteLine("Bye Bye!");
```

When the method has finished, it will return to the position from which it was invoked. The program flow when invoking a method, as in line 14 of Listing 3.1, is shown in Figure 3.8. The following individual steps can be identified in the figure. The numbers correspond to those shown in the figure.

1. Execute statements prior to line 14.
2. Execute line 14.
3. Call `System.Console.WriteLine` with the argument "Bye Bye!".
4. Execute the statements inside `System.Console.WriteLine(...)`.
5. Return to statement just after line 14.
6. Execute the rest of the statements in the `Main` method.

FIGURE 3.8

Program flow when invoking a method.



Line 10 contains another method call to `WriteLine`

10: `System.Console.WriteLine("Type y for yes; n for no. Then <enter>");`

The result of this method call is as follows:

`Type y for yes; n for no. Then <enter>.`

is printed onscreen, and the cursor is moved down one line.



Message: Another Term for Method Invocation

Consider line 9 of Listing 3.1. It contains a statement invoking the `WriteLine` method. Another term is often used in object-oriented programming circles to denote the invocation of an object.

When a method of object A contains a statement that is invoking the method of an object called B, we say A is sending a message to B. In line 10, our class `Hello` is sending a message to the `System.Console` class. The message is

`WriteLine("Type y for yes; n for no. Then <enter>")`

In general, objects are thought of as performing actions triggered by messages received. In our example, the action taken is to print

`Type y for yes; n for no. Then <enter>`

on the console.

Assigning a Value to a Variable

In line 11, we again reuse the `System.Console` class. This time, we use another of its `static` methods called `ReadLine`, which will pause the execution and wait for a response from the user. The response must be in the form of text entered and the Enter key pressed. As its name suggests, `ReadLine` will read the input from the user.

```
11:         answer = System.Console.ReadLine();
```

When the Enter key is pressed, the text typed in by the user will be stored in the `answer` variable. Consequently, if the user types ‘y’, `answer` will contain a “y”. If the user types ‘n’, `answer` will contain an “n”. This is because of the equals sign (=) placed after `answer`.

The equals sign (=) is used differently in C# than in standard arithmetic. In arithmetic, the equals sign usually denotes the equality of items to the left and right of the sign. For example, $4=2+2$ is a valid expression in arithmetic and can be said to be true. In C#, the meaning is quite different. Instead, the equals sign says “Make `answer` equal to `System.Console.ReadLine()`” or, in other words, “Store the text read from the keyboard in `answer`.”

The mechanism of giving `answer` a new value is called *assignment*. The text typed in by the user is said to be assigned to the `answer` variable. Line 11 is called an *assignment statement* and the equals sign (=) is called an *assignment operator* when used in this context. The equals sign is involved in other contexts and then it will have different appropriate names.



An Advantage of Declaring Variables

Initially, variable declarations seem to complicate matters and be a waste of typing and time. Why can’t we just use the variable when needed without any previous declaration? Well, in extremely short programs, such as the one presented, we could easily live without declarations. However, variable declarations have important benefits when writing larger programs. Consider a programming language like an older version of BASIC (not Visual Basic) where variable declarations are not required. Without further ado, BASIC allows us to involve, say, `MyVariable` in an assignment statement as in the following:

```
MyVariable = 100
MyVarable = 300
System.Console.WriteLine(MyVariable)
```

Initially, we stored `100` in `MyVariable` and later wanted to store `300` instead. However, due to a spelling mistake in the second line shown, we ended up storing `300` in `MyVarable`, leaving `MyVariable` with the same value 100. This mistake was not picked up by the BASIC compiler, but would have been unveiled by the C# compiler, where all variables used throughout the program are checked for matching declarations.

Now, try and put your trusty C# compiler to the test. Change the spelling of `answer` in line 11 or 12 and observe its reaction when you compile the now faulty program.

Branching with the `if` Statement

You saw one use for the equals sign in line 11. In line 12, it is used in a different context, this time resembling that of arithmetic’s standard use of the equals sign.

```
12:         if (answer == "y")
13:             System.Console.WriteLine("Hello World!");
```

C# uses two adjacent equals signs (==), called an *equality operator*, to denote a comparison between what is on its left with what is on its right. Consequently, the two following expressions articulate the same question:

Using standard arithmetic:

$2 + 3 = 6$ meaning “Is $2 + 3$ equal to 6?”

Answer: false

Is the same as

Using the C# language

`2 + 3 == 6` meaning “Is $2 + 3$ equal to 6?”

Answer: `false`

In line 12, we ask the question `answer == "y"`, meaning “Is `answer` equal to “y”?” The answer can be either `true` or `false`.

An expression that can only have the two values—`true` or `false`—is said to be a *Boolean expression*.



Caution

It is easy to confuse the equality operator == with the assignment operator =. The assignment operator = can be read “gets the value of” or “gets,” whereas the equality operator == should be read “is equal to.” To avoid confusion, some programmers refer to the assignment operator as *equals equals* or *double equals*.

By putting the keyword `if` in front of the Boolean expression `answer == "y"` and surrounding this expression by parenthesis, we are saying that only if `answer == "y"` is `true` should the line immediately after line 12 (in this case line 13) be executed. Conversely, if `answer == "y"` is `false`, jump over line 13 and continue with line 14. In Figure 3.9 I have zoomed in on lines 12–14 and indicated the execution flow with arrows.

Lines 12 and 13 contain a vital element for controlling the output of the computer and, in fact, achieve two different outputs as shown previously in sample output 1 and 2.

Note a couple of important details:

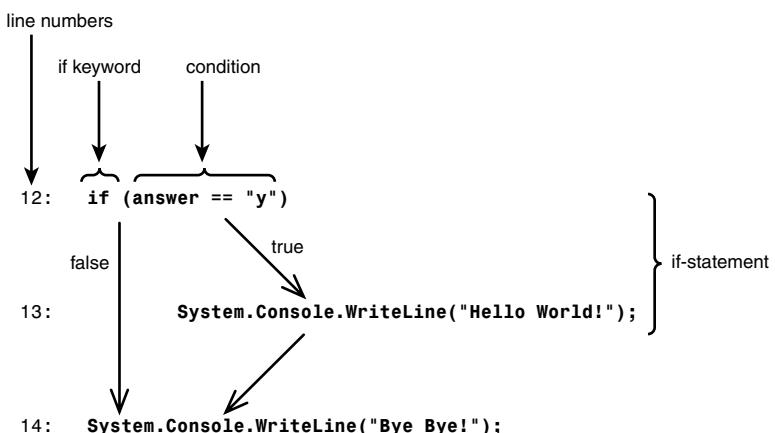
- Regardless of whether we answer `y` or `n`, the program writes the line “Bye Bye!” at the end.

- Only the answer `y` will trigger the program to print `Hello World!`. Any other answer, such as `horse`, `Peter`, `Y`, `yes`, `n`, `N`, and so on will make the Boolean expression false and ignore line 13.

Line 12 and 13 constitute an `if` statement. The `if` statement is said to control the flow of the execution because the execution at this point can choose to follow two different directions. The `if` statement is said to belong to a category of statements called branching statements.

FIGURE 3.9

The flow of execution in an `if` statement.



Caution

The left and right parenthesis enclosing the condition of the `if` statement, as shown in the following:

```
if (answer == "y")
```

are always required. Omitting them will produce a syntax error.

Ending the `Main()` Method and the `Hello` Class

The `}` on line 15 ends the `Main` method block started in line 6.

```
15: }
```



Caution

The braces `{}` must occur in pairs. Failing to comply with this rule triggers a compiler error.

In line 16, `}` ends the `Hello` class definition block.

```
16: }
```

This concludes the analysis of Listing 3.1.

A Few Fundamental Observations

The previous sections were able to extract many fundamental C# constructs and mechanisms from Listing 3.1, despite its simple appearance. This section looks at a few general C# concepts by summarizing and building on the knowledge you have gained from Listing 3.1.

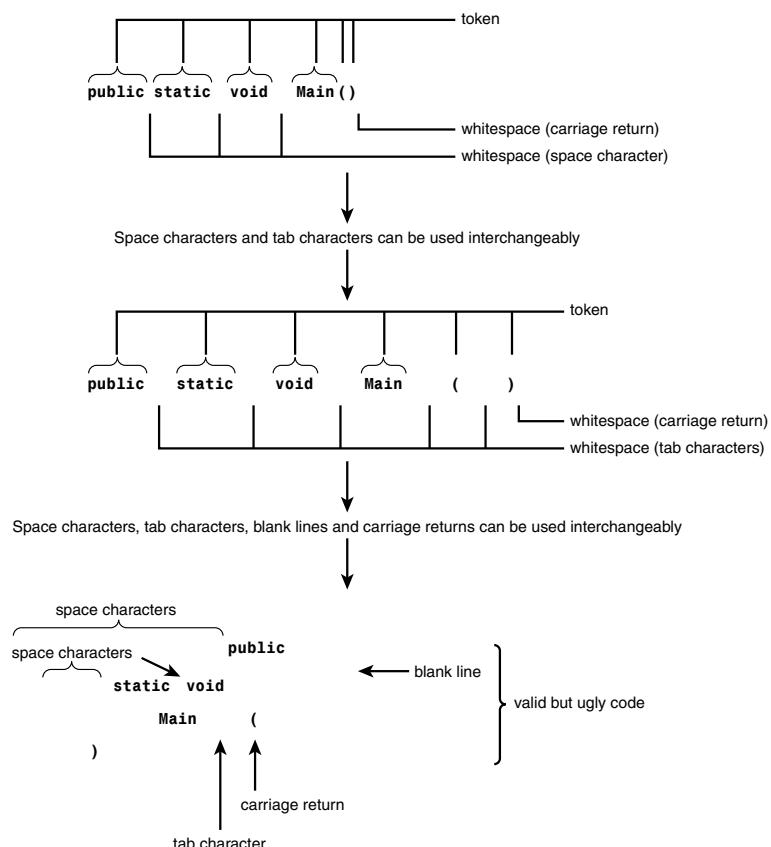
C#'s Source Code Format

Blank lines, space characters, tab characters, and carriage returns are collectively known as *whitespace*. To a large degree, the C# compiler ignores whitespace. Consequently, you can use blank lines, space characters, tab characters, and carriage returns interchangeably. Let's look at an example. From the C# compilers point of view, the three lines of source code in Figure 3.10 are all identical to line 5 of Listing 3.1.

```
05:     public static void Main()
```

FIGURE 3.10

Whitespace is ignored by the compiler.



Notice how distorted the third line in Figure 3.10 has become; it is still valid code but incomprehensible and ugly.

The indivisible elements in a line of source code are called *tokens*. One token must be separated from the next by whitespace, commas, or semicolons. However, it is unacceptable to separate the token itself into smaller pieces by using whitespace or separators.

A *token* is a particular instance of a word with a special meaning attached to it. The term token is also frequently used in the scientific fields of logic and linguistics.

Don't break up tokens with whitespace; this causes invalid code as shown below, which contains three invalid versions of line 5: `public static void Main():`

```
pub lic  stati c  void  Main()
public static void Ma
in()
public static vo id Main()
```

On the other hand, it is possible to aggregate statements and source code on the same line, as shown in the following line.

```
class Hello { public static void Main() { string answer;
```

The following is better-styled original code:

```
02: class Hello
03: {
04:     // The program begins with a call to Main()
05:     public static void Main()
06:     {
07:         string answer;
```

Although C# provides much freedom when formatting your source code, you can increase its clarity considerably by following a reasonable style. It is possible to write valid but ugly source code, as you have seen in a few of the previous examples. Ugly here refers to messy, unclear, confusing source code that is difficult for another person to comprehend.

Listing 3.1 follows a certain style that is adhered to by a large proportion of the programming community. Let's have a look at a few general guidelines. Line numbers from Listing 3.1 are provided as examples.

- Have one statement per line (lines 7, 9, 10, 11, and 14).

However, note the `if` statement; it's an exception. It should be spread over several lines (in the case of Listing 3.1 it is spread over two lines).

- After an opening brace (`{}`) move down one line and indent (lines 4, and 7).
- Indent matching pairs of braces identically (lines 3, 16 and lines 6, 15).
- While observing the two previous rules, indent lines between matching pairs of braces identically. (Lines 4, 5 and lines 7–14). Keep in mind that statements after `if` conditions should be indented (line 13).

- Put in blank lines to separate distinctive logic parts of the source code (line 8).
- Do not use whitespace around the parentheses associated with a function name (line 5).

Competent programmers apply other conventions, to improve the style of their source code. We will discuss these in the following chapters whenever relevant.

A Brief Tour Around the .NET Framework

We have already made extensive use of the .NET Framework class library in Listing 3.1. It's especially visible in lines 9, 10, 11, 13, and 14 with `System.Console.WriteLine` and `System.Console.ReadLine`. There is extensive support for equipping your program with a wide variety of functionality from this class library. But how do we know which particular classes and methods we have access to and can reuse and how do we know their characteristics and the functionality they provide? We can find answers to these questions in the comprehensive documentation provided for the class library. The simple aim of this section is to show you how to locate this documentation and confirm the existence of `System.Console.WriteLine` and `System.Console.ReadLine` inside the myriad of other classes and methods.

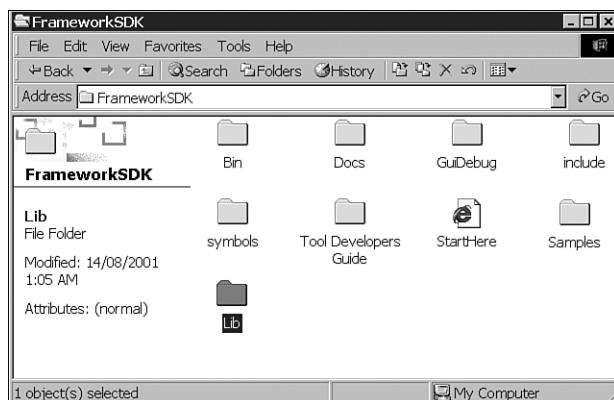
At the time of writing, the following set of commands can be used to locate the .NET Framework documentation. This might change in later versions of the Software Development Kit (SDK).

Open up the directory with the path `D:\Program Files\Microsoft .Net\FrameworkSDK`.

Notice that `D:\` could be another letter, depending on where your documentation has been installed.

You should then see a window displaying the contents of the FrameworkSDK folder, similar to that shown in Figure 3.11.

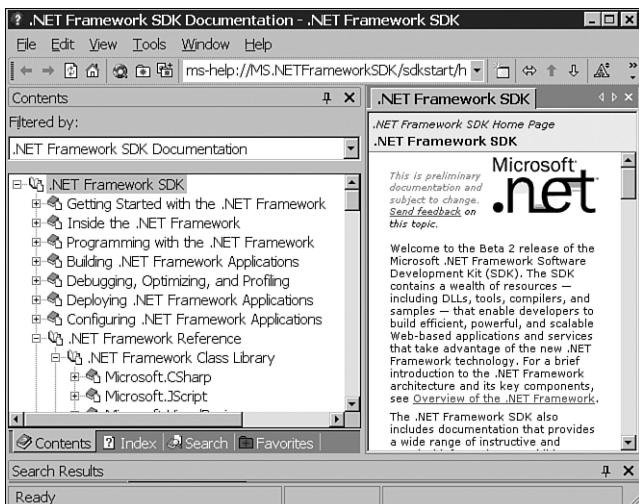
FIGURE 3.11
Contents of the
FrameworkSDK folder.



Double-click the `StartHere` icon to bring up the start page of the .NET Framework Reference. Display the start page of the .NET Framework Documentation by clicking the hyperlink called

.NET Framework SDK documentation situated in the Documentation section. Expand the appearing .NET Framework SDK node on the left hand side of the window. Locate the .NET Framework Reference node and expand it. Among many other appearing nodes one is called .NET Framework Class Library which you can expand to display a window similar to that shown in Figure 3.12.

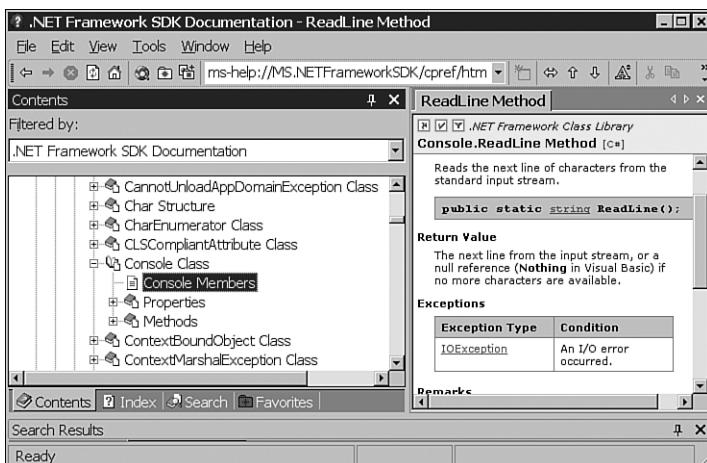
FIGURE 3.12
The .NET Framework
Class Library
Documentation.



You are now free to browse the documentation for the class library.

To locate the `Console` class, expand the System node and scroll down to find and expand the `Console` Class node. You can now find the `WriteLine` and `ReadLine` methods by clicking the `Console Members` node. Click the `ReadLine` hyperlink in the right part of the window to see the window shown in Figure 3.13.

FIGURE 3.13
Displaying the
`Console.ReadLine` specifi-
cations.



Under the headline `Console.ReadLine` Method, you can read a short description of `ReadLine`. You can locate the `WriteLine` method in the same fashion.

Many of the terms mentioned in the .NET Framework Reference will not make any sense to you right now. However, eventually this reference is most likely going to be an important source of information.

As you are presented with more classes from the library, try to locate them and use the new terms introduced in the coming chapters to familiarize yourself with this very powerful set of tools.

C# Statements

As mentioned earlier, each method in C# contains a collection of statements. Many types of statements exist in C#. The following is a brief summary of the statements we encountered in Listing 3.1.

- Declaration statements (line 7)
- Assignment statements (line 11)
- Method call statements (lines 9, 10, 11, 13, and 14)

Declaration statements create a variable that can be used in the program. They announce the type and identifier of the variable.

Even though declaration statements may seem superfluous, they introduce several features that help eliminate bugs in computer programs.

An assignment statement uses the assignment operator `=`. It assigns a value to a storage location that is represented by the identifier of the variable.

A method call activates a method. It can send arguments to the invoked method that utilizes these to perform its actions. When the invoked method terminates, the program will return to the statement immediately following the method call statement.

Summary

In this chapter, you have learned about abstraction and encapsulation, two important object-oriented concepts. You have also been presented with a simple C# program and learned about its basic C# elements and general underlying concepts.

The following are the important points covered in this chapter:

Abstraction and encapsulation are important concepts in object-oriented programming.

Abstraction allows programmers, through simplification, to cope with complexity. Only attributes relevant to the problem at hand should be included in an abstraction.

Encapsulation packs data and their associated actions into the class entity. Only relevant parts of an object are exposed to the outside world. This keeps the data inside an object.

non-corrupted and provides for a simpler interface to the outside world. `public` and `private` are two important C# keywords used when implementing the encapsulation concept.

The method of one object can call the method of another object. In OO terminology, we say that a message is sent between two objects.

A class is written in the source code and is an inactive blueprint for its dynamic object counterparts that are created inside the main memory during the execution of a program.

Comments are ignored by the compiler but are used to make the code clearer for the person reading through the source code.

The keywords in C# were chosen by C#'s designers so their names and predefined meaning never change. In contrast, identifiers vary between programs, are decided by the programmer, and are used to name C# constructs, such as classes, methods, and instance variables.

A block forms a logical unit in a program. Among other functions, blocks are used to indicate which parts of a program belong to which classes and methods.

Every program must have one `Main` method. The `Main` method is called by the .NET runtime and is the first part of a program to be executed.

A variable has a name (identifier), is of a specific type, and represents a memory location containing a value.

A variable of type `string` can be used to store text.

Any set of actions performed by C# can be broken down into simple instructions called statements.

The predefined functionality in the .NET Framework class library can be conveniently accessed from the C# source code.

Methods are defined inside classes and contain statements. When a method is called, its statements are executed in the same sequence as they are written in the source code.

The equals sign is used in two different fashions, as an assignment operator (`=`) and as an equality operator (`==`).

The `if` statement is able to change the program's flow of execution. The path followed depends on the Boolean value of its condition.

To make the C# source code clear for the reader, follow a certain format and style.

The .NET Framework class library has comprehensive documentation attached.

Three common statements found in C# are declaration statements, assignment statements, and method call statements.

Review Questions

1. How does abstraction help the programmer to cope with complexity?
2. Is the idea behind encapsulation confined to software design? Give an example from everyday life.
3. What are the advantages of using encapsulation in software construction?
4. Which two C# keywords are important for implementing encapsulation?
5. What are the differences between a class and its objects?
6. What is the significance of the class interface?
7. How do you specify the beginning of a comment?
8. Why use comments if the compiler ignores them?
9. What are keywords and identifiers?
10. How is a block specified in C#? What are blocks used for?
11. Can you write a program without a `Main` method? Why or why not?
12. What are the essential parts of a variable?
13. What is a simple C# instruction called? How is it terminated?
14. Which class and which method can you call in the .NET Framework class library to print text on the console? Write a statement that prints “My dog is brown.”
15. How is a method called? What happens when a method is called?
16. What is an assignment? Which symbol is used to perform an assignment?
17. What is the advantage of declaring variables?
18. How can you make the program decide between two paths of execution?
19. What is whitespace? Does the compiler care much about whitespace?
20. Do all programmers have to follow the same style to write valid C# programs?

Programming Exercises

In the following exercises, you are meant to change and add parts to the program in Listing 3.1 to make it perform the suggested actions.

1. Instead of printing “Bye Bye!” as the last text on the command console before the program finishes, change the source code so that the program writes “Bye Bye. Have a good day!”

2. Instead of typing a **y** to have the program print “Hello World!”, have the user type **Yes**. Have the program inform the user about this by changing line 10.
3. Instead of using the variable name **answer** to hold the input from the user, change the name to **userInput**.
4. Let the program print out an additional line under “**Bye Bye. Have a good day!**” saying “**The program is terminating.**”
5. Declare another variable of type **string** called **userName**. Before the program prints “Do you want me to write the famous words?”, have the program request the user to type in his or her name. After the user has entered his or her name, have the program read this name and store it in the **userName** variable. Then have the program print “Hello” followed by the content of the username. Tip: the last printout can be accomplished by typing

```
System.Console.WriteLine("Hello" + userName);
```

A typical execution of the program should result in the following output:

```
Please type your name
Deborah<enter>
Hello Deborah
Do you want me to write the famous words?
Type Yes for yes; n for no. Then <enter>.
Yes<enter>
Hello World!
Bye Bye. Have a good day!
The program is terminating.
```

CHAPTER 4

A GUIDED TOUR THROUGH C#: PART II

You will learn about the following in this chapter:

- More fundamental C# elements building on those presented in Chapter 3, “A Guided Tour Through C#: Part I,” through a source code driven presentation
- The namespace concept and an understanding for how this feature can help organize and access C# components
- The `int` type for storing whole numbers
- How to write and call your own user-defined methods
- Multiline comments and their different styles
- How C# evaluates simple expressions
- How methods can be used to simplify your source code

Introduction

A new source code example containing a simple calculator is presented in this chapter, along with several new elements of the C# language.

Essential Elements of SimpleCalculator.cs

You have already met the `string` type in Chapter 3, “A Guided Tour through C#: Part I.” The calculator program you will see shortly introduces another type called `int`, which is a commonly used type for variables holding whole numbers. You will further see how you can define and use your own methods. The important namespace concept, enabling classes and other types to be organized into a single coherent hierarchical structure while providing easy access, is also presented. `WriteLine` will show off by revealing its versatility, and you will finally meet the handy `Math` class and apply a couple of methods from this part of the .NET Framework class library.

Presenting SimpleCalculator.cs

The source code in Listing 4.1 calculates the sum, product, minimum, and maximum value of two numbers entered by the user. The program prints the answers on the command console.

LISTING 4.1 Source Code for SimpleCalculator.cs

```
01: using System;
02:
03: /*
04:  * This class finds the sum, product,
05:  * min and max of two numbers
06: */
07: public class SimpleCalculator
08: {
09:     public static void Main()
10:     {
11:         int x;
12:         int y;
13:
14:         Console.Write("Enter first number: ");
15:         x = Convert.ToInt32(Console.ReadLine());
16:         Console.Write("Enter second number: ");
17:         y = Convert.ToInt32(Console.ReadLine());
18:         Console.WriteLine("The sum is: " + Sum(x, y));
19:         Console.WriteLine("The product is: " + Product(x, y));
20:         Console.WriteLine("The maximum number is: " + Math.Max(x, y));
21:         Console.WriteLine("The minimum number is: " + Math.Min(x, y));
22:     }
23:
24:     // Sum calculates the sum of two int's
25:     public static int Sum(int a, int b)
26:     {
27:         int sumTotal;
28:
29:         sumTotal = a + b;
30:         return sumTotal;
31:     }
32:
33:     // Product calculates the product of two int's
34:     public static int Product(int a, int b)
35:     {
36:         int productTotal;
37:
38:         productTotal = a * b;
39:         return productTotal;
40:     }
41: }
```

The following is the sample output when the user enters **3** and **8**:

```
Enter first number: 3<enter>
Enter second number: 8<enter>
```

```
The sum is: 11
The product is: 24
The maximum number is: 8
The minimum number is: 3
```

A quick reference to the source code in Listing 4.1 is provided in Listing 4.2. You will most likely recognize several constructs from `Shakespeare.cs` in Chapter 2, “Your First C# Program,” and `Hello.cs` in Chapter 3, and you will probably begin to form a picture of what the fundamental elements of a typical C# program look like.

LISTING 4.2 Brief Analysis of Listing 4.1

```

01: Allow this program to use shortcuts
    ↵when accessing classes in System namespace
02: Empty line
03: Begin multi-line comment
04:     Second line of multi-line comment: This class finds the sum, product,
05:     Third line of multi-line comment: min and max of two numbers
06: End multi-line comment
07: Begin the definition of a class named SimpleCalculator
08: Begin the block of the SimpleCalculator class definition
09:     Begin the definition of a method called Main()
10:     Begin the block of the Main() method definition
11:         Declare a variable called x which
            ↵can store whole numbers (integers).
12:         Declare a variable called y which
            ↵can store whole numbers (integers).
13:     Empty line
14:     Print out: Enter first number:
15:     Store users answer in the x variable. Move down one line
16:     Print out: Enter second number:
17:     Store users answer in the y variable. Move down one line
18:     Print out: The sum is: followed by the
            ↵value returned from the Sum method.
19:     Print out: The product is: followed by the
            ↵value returned from the Product method.
20:     Print out: The maximum number is: followed by the
            ↵value returned by the Max method of the Math class.
21:     Print out: The minimum number is: followed by the
            ↵value returned by the Min method of the Math class
22: End the block containing the Main() method definition
23: Empty line
24: Make comment: Sum calculates the sum of two int's
25: Begin the definition of a the method Sum(int a, int b)
26: Begin the block of the Sum(int a, int b) method definition
27:     Declare a local variable called sumTotal which
            ↵can store whole integers.
28:     Empty line
29:     Find the sum of a and b; store this result in sumTotal
30:     Terminate the method; return the value of sumTotal to the caller.
31: End the block of the Sum(int a, int b) method
32: Empty line.
```

LISTING 4.2 continued

```

33:    Make comment: Product calculates the product of two int's
34:    Begin the definition of a the method Product(int a, int b)
35:    Begin the block of the Product(int a, int b) method definition
36:        Declare a local variable called productTotal which
            ➔can store whole integers
37:        Empty line
38:        Find the product of a and b; store this result in productTotal.
39:        Terminate the method; return the
            ➔value of productTotal to the caller
40:    End the block of the Product(int a, int b) method
41: End block containing SimpleCalculator class definition

```

By now, I assume you are comfortable writing source code in Notepad and compiling it with the `csc` command of the command console; therefore, I will let you write and run the source code in Listing 4.1 in peace now.

A Closer Look at `SimpleCalculator.cs`

`SimpleCalculator.cs` is probably not a program people would rush to buy in their local software store, but it certainly contains a few essential ingredients that would be part of most C# best-selling computer programs. The following section will take a closer look at Listing 4.1 to uncover them.

Introducing Namespaces

We need to introduce the namespace concept to get an initial understanding for line 1. To this end, we move to a seemingly unrelated place—your home.

01: using System;

Have you ever, like me, had the feeling of an invisible spirit at play that constantly turns your home into a place resembling a test site for new explosives, despite your best efforts to the contrary?

There seems to be a large number of different objects in a home causing the mess to relentlessly get out of hand—shirts, spoons, knives, shampoo, books, CDs, apples, pots, and Picasso paintings all pile up.

How, then, do we manage to make our home neat and tidy before the family arrives at Christmas? Containers are the solution to our problem. We form small hierarchies of containers inside other containers holding similar kinds of objects. For example, a kitchen knife (the object) would be positioned in the kitchen (here regarded as being a container) in the upper-left drawer container, in the cutlery tray container, in the knife compartment container, together with other more-or-less similar knives.

Not only do these containers help us to make our home nice and tidy, they also allow us to get an overview of where different objects are stored in the house. We are even able to show our guests where to find different objects. When everything is tidy around Christmas time, we can confidently tell our new friend Fred how to find a knife. If we turned our home into a small hotel with new visitors arriving on a daily basis, we might further create a little system for how

to tell new guests where to find different objects. In this referencing system, we could decide to simply put a dot (period) between each container name. A guest looking for a knife then might only need the message `Kitchen.UpperLeftDrawer.CutleryTray.KnifeCompartment`.

As a final bonus, we avoid name collision problems. For example, we can differentiate between a knife used for fishing found in the garage and a knife used for eating found in the kitchen, even though they are both referred to as knife. This is simply done by specifying where each particular knife is positioned. Accordingly, the kitchen knife specified by `Kitchen.UpperLeftDrawer.CutleryTray.KnifeCompartment.Knife` would likely be different from our fishing knife referenced by `Garage.FishingCupboard.UpperRightDrawer.KnifeBox.Knife`.

The kitchen contains not only drawers and boxes, it also can contain objects like chairs and tables. Similarly, every container can contain not only other containers but also objects.

The practical container system described here is similar in concept to the namespace idea. The namespaces act as containers for the many classes we construct and must keep track of. Namespaces help us organize our source code when developing our programs (keeping our “home” of classes “nice and tidy,” allowing us to know where each class is kept). They also allow us to tell our “guest” users where classes are kept, so they (other programmers) easily can access and reuse our classes. Name collisions between classes created by different programmers, perhaps from different software companies, are eliminated, because each class can be uniquely referenced through its namespace name.

Recall the .NET Framework documentation viewed previously. If you had a good look, you would have encountered a myriad of classes. In fact, the .NET Framework contains several thousand classes. Consequently, the .NET Framework is heavily dependent on namespaces to keep its classes organized and accessible.

The .NET Framework contains an important namespace called `System`. It holds specific classes fundamental to any C# program. Many other `System` classes are frequently used by most C# programs. It also includes our familiar `Console` class used extensively in Listing 3.1 (lines 9–11 and 13–15) of Chapter 3 and lines 14–21 in Listing 4.1.

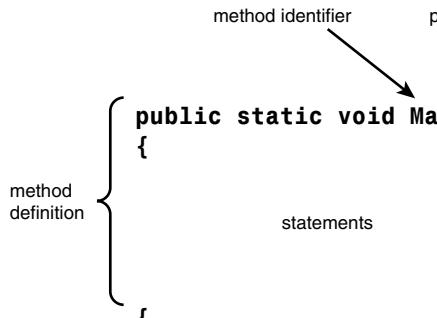


Tip

Avoid giving names to classes that are identical to important, often-used namespace identifiers, such as `System`, `Collections`, `Forms`, and `IO`.

Listings 3.1 and 4.1 represent two different options available when accessing classes of the `System` (or any other) namespace in your source code:

- Without `using System;` in the source code, as in line 1 of Listing 3.1—We must then reference the `System` namespace explicitly every time we use one of its classes, as in Listing 3.1, line 14:



- With `using System;`, as in line 1 of Listing 4.1—We can use any class of the `System` namespace without the repetitive tedious chore of explicitly typing `System`. The previous example of code could then be truncated to the following (just like lines 14–21 of Listing 4.1).

```

Console.WriteLine("Bye Bye!");
  
```

Annotations explain the code:

- An upward arrow labeled "System namespace reference (removed and not required)" points to the word "Console".

`using` essentially frees us from the burden of writing fully-qualified namespace names in numerous places of our source code. It also enhances the readability of our code by abbreviating otherwise lengthy references.



Note

`using` can only be applied to namespaces and not classes, so the following line is invalid:

```

using System.Console; ← invalid
  
```

Annotations explain the invalid code:

- An upward arrow labeled "class" points to the word "Console".
- A brace under "Console" is labeled "class".
- An arrow labeled "invalid" points to the word "Console".

Multi-Line Comments

You have already met the double slash `//` used for single line comments in lines 1 and 4 of Listing 3.1. `/*` (forward slash star) indicates the beginning of a multi-line comment. Anything written between `/*` and the matching `*/` is a comment and is ignored by the compiler (as in line 3 of Listing 4.1).

03: `/*`

// could be used for multi-line comments, but you would have to include // in every line of the comment, as shown in the following:

```
//This comment spans
//over two lines
```

/* */ could likewise be used for a single line comment:

```
/* This is a single line comment */
```

Most programmers seem to prefer // for one or at most only a few lines of comments, whereas /* */ facilitates elaborate comments spanning numerous lines.

A few different multi-line commenting styles are often seen. The start comment /* and end comment */ have been highlighted so it is easier for you to locate the beginning and end of each comment.

```
/* Many attempts to communicate
   are nullified by saying too much.
```

Robert Greenleaf */

```
/*
 * I have made this rather
 * long letter because I haven't
 * had the time to make it shorter.
 *
 *        Blaise Pascal
 */
```

```
*****
 * It's a damn good program.
 * If you have any comments,
 * write them on the back
 * of the cheque.
 *
 *     Adapted from Erle Stanley Gardner
*****
```

Probably not the typical content of a comment, but these comments illustrate three different popular multi-line commenting styles and possess wisdom applicable when commenting source code.

The actual text of the multi-line comment is seen in lines 4 and 5 of Listing 4.1:

```
04: * This class finds the sum, product,
05: * min and max of two numbers
```

In line 6, */ matches /* of line 3 and terminates the multi-line comment.

```
06: */
```



Tip

Making good comments is like adding salt to your food; you have to add the right amount. Excessive use of comments obscures the readability of the code and gives the reader more to read than necessary. It is as damaging as too few comments.

Rather than merely restating what the code does by using different words, try to describe the overall intent of the code.

Coding nightmare:

```
/*
 * The next line adds s1 to s2. s3 is then
 * again added to the previous result and finally s4 is
 * added to the latter result. This result is then stored in
 * the ss variable. s1, s2, s3 and s4 all represent different
 * speeds of a car. ss is the sum of those speeds.
 */
ss = s1 + s2 + s3 + s4; ← Inferior choice of identifiers
```

Poor comment. It only repeats in a verbose fashion what is expressed very precisely in C# in the next line.

```
/*
 * The next line takes the ss variable from above and divides
 * it by 4 it then stores this result in the variable a which
 * then is the average of s1, s2, s3 and s4.
 */
```

```
a = ss / 4;
```

Bad identifier choice.

Another poor comment.

Let's see how we can improve this source code:

Improved code 😊

```
the comment states the intent of the code briefly
//Calculate the average speed of a car
```

```
speedSum = speed1 + speed2 + speed3 + speed4;
averageSpeed = speedSum / 4;
```

Good choice of identifiers makes the code self documenting

Declaring a Variable of Type `int`

Line 11 contains a statement declaring a variable called `x`. Remember how we previously declared a variable of type `string` in Listing 3.1 line 7, enabling it to contain a string of characters (text). This time, we have exchanged the keyword `string` with the keyword `int`. It dictates the type of `x` to be a specific kind of integer taking up 32 bits of memory (4 bytes).

```
11:     int x;
```

Integers, in general are whole numbers, such as 8, 456, and -3123. This contrasts with *floating-point* numbers that include a fraction. 76.98, 3.876, and -10.1 are all examples of floating point numbers. Because `x` now takes up 32 bits of memory, it can represent numbers in the -2147483648–2147483647 range. We will not examine this fact closely now, but observe the following calculation:

$$2^{32}/2 = 2147483648$$

The first 2 is the bit size, the 32 is the number of bits, and the `/2` shows that half of the available numbers are allocated for positive numbers, and half for negative numbers.

Binary numbers are discussed at length in Appendix D, “Number Systems,” which is found on www.samspublishing.com, while integers and their range will be discussed thoroughly in Chapter 6, “Types Part I: The Simple Types.”

Line 12 declares a variable called `y` to be of type `int`.

```
12:     int y;
```

`x` and `y` are usually considered to be unacceptable names for variables because identifiers must be meaningful and reflect the content of the variable. In this particular case, however, `x` and `y` are involved in generic arithmetic calculations and do not represent any particular values, such as average rainfall or the number of newborns in Paris. Any math whiz kid would immediately accept `x` and `y` as valid name choices in a mathematical equation with equivalent generic characteristics.

Converting from Type `string` to Type `int`

Any user response typed on the console, followed by Enter, and received by `Console.ReadLine()` is considered to be of type `string` by the C# program. Even a number like 432 is initially presumed to merely be a set of characters and could easily be ABC or #@\$ instead.

A number represented as a `string` cannot be stored directly in a variable of type `int`; so we must first convert the `string` variable to a variable of type `int` before commencing.

By using the `ToInt32` method of the `Convert` class as in line 15, the program attempts to convert any user input to an `int` number. After successful conversion, the part on the right side of the equals sign in line 15 will represent the corresponding `int` number, and be stored in `x`. However, the conversion can only take place if the user input matches an `int` number. Thus, inputs such as **57.53** or **three hundred** will trigger an error, whereas **109** or **64732** are accepted.

```
15:     x = Convert.ToInt32(Console.ReadLine());
```

Creating and Invoking Your Own Methods

Before line 18 can be understood properly, we need to jump ahead and look at lines 25–31.

```
25:     public static int Sum(int a, int b)
26:     {
27:         int sumTotal;
28:
29:         sumTotal = a + b;
30:         return sumTotal;
31:     }
```

Until now, we have happily used pre-fabricated methods, such as `Console.ReadLine()` and `Console.WriteLine()`. Despite the vast array of functionality found in the .NET Framework and other available commercial class libraries, you will, when creating new unique source code, need to supply specialized functionality by writing your own user-defined methods.

Lines 25–31 are an example of a user-defined method called `Sum`.

Briefly explained, `Sum`, when called, receives two numbers. It adds the two numbers together and returns the result to the caller.

The definition for the `Sum()` method, with its method header, braces {}, and method body, follows the same general structure as the definition for the `Main` method.



Tip

Generally, classes represent objects, whereas methods represent actions. Try to adhere to the following when you name classes and methods:

Use nouns (`Car`, `Airplane`, and so on) when naming classes.

Use verbs for method names (`DriveForward`, `MoveLeft`, `TakeOff`, and so on).

The method header of line 25 is, in many ways, similar to the method header of `Main` in line 9 (recall the access modifier `public` and its ability to let a method be included in the interface of the class to which it belongs).

We have also previously introduced `static`, in line 5 of Listing 3.1, and will abstain from any further description of this keyword right now.

However, `void` has been replaced by `int`, and we see what looks like two variable declarations separated by a comma residing inside a pair of parentheses (`int a, int b`) located after `Sum`. This deserves an explanation.

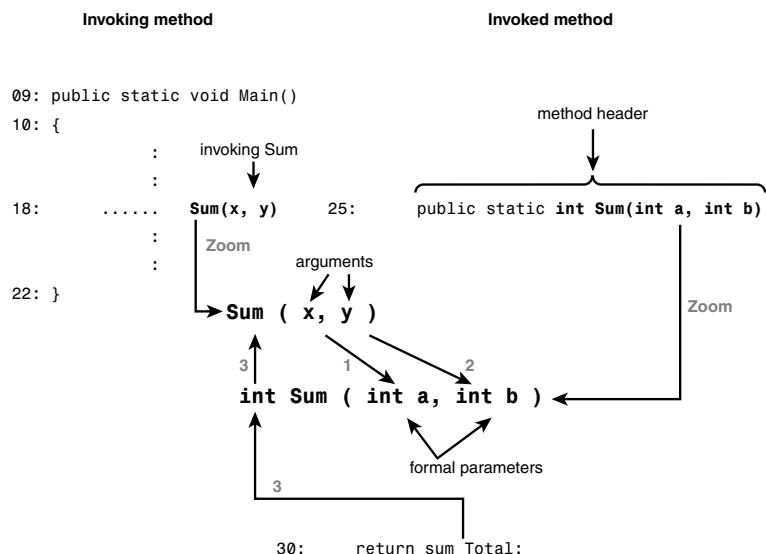
Line 25 can be regarded as the interface of the `Sum` method. This interface facilitates the communication between the method calling `Sum` (in this case `Main`) and the method body of `Sum`.

In everyday language, an *interface* can be described as a feature or circumstance that enables separate (and sometimes incompatible) elements to communicate effectively.

Figure 4.1 zooms in on the process involved when `Sum(int a, int b)` of line 25 is invoked from line 18 `Sum(x, y)` of the `Main()` method. The two arrows marked Zoom simply indicate which parts have graphically been blown up and zoomed in on.

FIGURE 4.1

Invoking a user-defined method.



a and **b** in `Sum`'s method header are called formal parameters. A *formal parameter* is used in the method body as a stand-in for a value that will be stored here when the method is called. In this case, the values of the arguments **x** and **y** of line 18 will be plugged into **a** and **b** indicated with arrows 1 and 2, just as though the two assignment statements `a=x;` and `b=y;` were executed.

a and **b** can now be used inside the method body (in this case line 29) just like any regular declared variable initially holding the values of **x** and **y**.

The `int` in line 25 replacing `void` not only tells us that `Sum` returns a value of some sort, but also that this value is of type `int`. This leads us to line 30, where we find the C# keyword `return`.

```
30:         return sumTotal;
```

`return` belongs to a type of statement called a return statement. When a `return` statement is executed, the method is terminated. The flow of execution is returned to the caller and brings along the value residing in the expression after `return` (in this case the value of `sumTotal`. See the arrows with number three in Figure 4.1).

Accordingly, the type of `sumTotal` must be compatible with the return type specified in the method header (in this case `int`), as illustrated in Figure 4.2.



No Data Are Returned from a *void* Method

When the keyword `void` is positioned in front of the method name in the method header, it indicates that when the runtime has finished executing the statements of the method, it will not return any data to the caller. Consequently the `void` keyword has the opposite meaning from the keyword `int` when put in front of the method name in the method header. The latter indicates that the method will always return a value of type `int`.

FIGURE 4.2

Return type of method header must match type of return expression.

```

25:     public static int Sum(int a, int b)
26:     {
27:         int sumTotal;
28:
29:         sumTotal = a + b;
30:         return sumTotal; ...sumTotal is returned to invoking method
31:     }           return statement

```

**Tip**

When you program in C#, think in terms of building blocks. Classes, objects, and methods are meant for this approach. Break down your programs into manageable building blocks. In a C# program, you typically use classes and methods that are

- From class libraries (NET Framework or other commercially available packages). Many class libraries can be obtained for free over the Internet.
- Written by yourself and, as a result, contain user-defined classes and methods.
- Created by other programmers, perhaps your fellow students or colleagues at work.

Don't attempt to reinvent the wheel. Before throwing time and money into creating new classes and methods, check to see if equivalent functionality has already been implemented, tested by professional programmers, and made available in the form of highly efficient and robust software components.

An Assignment Statement

Line 29 is an assignment statement. **a** and **b** are added together using the **+** operator. The result is stored in the **sumTotal** variable due to the assignment operator **=** (see Figure 4.3). Let's illustrate this by using the sample output example of from Listing 4.1, where the input provided was **3** and **8**. When **Sum** is called, **3** and **8** are transferred to **a** and **b**, through **x** and **y** of **Main**. After the execution of line 29, **sumTotal** holds the value **11**.

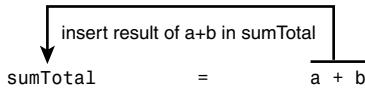
```
29:     sumTotal = a + b;
```

Combining Strings and Method Calls

In line 18, we first need to focus on the **Sum(x, y)** part, which, as already shown, invokes the **Sum** method defined in lines 25–31 and sends the two arguments **x** and **y** along. Notice that we do not need to include *ObjectName* or *ClassName* as specified in the previously given conventions, *ObjectName*.*MethodName*(*Optional.Arguments*) and *ClassName*.*MethodName*(*Optional.Arguments*), because the invoker and the invoked method reside inside the same class—**SimpleCalculator**.

FIGURE 4.3

Assigning the result of a calculation to a variable.



```
18:     Console.WriteLine("The sum is: " + Sum(x, y));
```

After the `Sum` method returns, you can virtually substitute `Sum(x, y)` with `sumTotal` from line 30. In our example, `Sum(x, y)` will represent the value 11 (see Figure 4.4).

FIGURE 4.4

`Sum(x, y)` represents the value of `sumTotal` when `Sum` returns.

```
18:     Console.WriteLine("The sum is: " + Sum(x, y));
```

```

25:     public static int Sum(int a, int b)
26:     {
27:         int sumTotal;
28:
29:         sumTotal = a + b;
30:         return sumTotal;
31:     }
  
```

`Sum(x, y)` represents the value of `sumTotal`

Even though `Sum(x, y)` resides inside another method call in the form of `Console.WriteLine()`, C# can easily cope and correctly resolves the sequence of events that need to take place. When the value of `Sum(x, y)` is needed, the call will be made. Line 18 could then, after lines 25–31 have been executed and the flow has returned to line 18, be thought of as:

```
Console.WriteLine("The sum is: " + 11);
```

Because 11 is inside the parentheses of a call to `WriteLine`, it is automatically converted to a `string`. Consequently, we can now look at line 18 as follows:

```
Console.WriteLine("The sum is: " + "11");
```

We still have one odd detail to sort out—the `+` symbol between `"The sum is: "` and `"11"`. When the `+` symbol is positioned between two numeric values, it adds them together in standard arithmetic fashion. However, when the C# compiler finds the `+` symbol surrounded by strings, it changes its functionality. It is then used to connect the two strings surrounding it, which results in `"The sum is: 11"`. In general, connecting or pasting together two strings to obtain a larger string is called *concatenation*, so the `+` symbol in this case is referred to as a *concatenation operator*.

Finally, line 18 has essentially been transformed to look like the following:

```
Console.WriteLine("The sum is: 11");
```

This is familiar to us and says, “Print out The sum is: 11 on the console.” Fortunately, this is exactly what we see when running the program.



The + Symbol: Adding or Concatenating?

The + operator can be used for different purposes. One moment it is used for a string concatenation; the next it is involved in adding numbers together. Due to this flexibility of the + operator, we say it is *overloaded*. Overloaded operators are very useful, but they come at a price; hard to trace bugs and mysterious results can sometimes appear.

For example, when + acting as a string concatenation operator is confused with + acting as an addition operator, the program produces unexpected results. Consider the following line of code:

```
Console.WriteLine("x + 8 = " + x + 8);
```

If x has the value 6, which output would you expect?

The actual output is x + 8 = 68, whereas many would have expected x + 8 = 14.

Why is that? Well first, "x + 8 = " + x is concatenated to "x + 8 = 6" which again is concatenated with 8 to form x + 8 = 68.

A correct result printing x + 8 = 14 can be produced with the following code:

```
"x + 8 = " + (x + 8)
```



KISS

The C# compiler is capable of resolving extremely complicated, nested, convoluted statements and programs. Programmers with a bit of experience sometimes attempt to show off by stretching this power to create some warped, weird, and intricate programs that nobody apart from themselves can understand. This kind of programming practice is acceptable if you want to have a bit of fun in your free time at home, but it has no place in properly constructed software.

Be humble! Follow the KISS principle—Keep It Simple Stupid.

Lines 34–40 are very similar to lines 25–31, the only differences being different method and variable names and, instead of calculating the sum, the **Product** method calculates the product of two numbers. Note that the asterisk (*) character is used to specify multiplication.

```
34:     public static int Product(int a, int b)
35:     {
36:         int productTotal;
37:
38:         productTotal = a * b;
39:         return productTotal;
40:     }
```

Lines 18 and line 19 are conceptually identical.

```
19:     Console.WriteLine("The product is: " + Product(x, y));
```

**Tip**

The order in which the methods of a class are defined is arbitrary and has no relation to when you can call those methods in your source code and when they will be executed. Accordingly, you could change the order in which the `Sum` and `Product` definitions appear in the definition of the `SimpleCalculator` class.

The `Math` Class: A Useful Member of the .NET Framework

Line 20 is similar in concept to lines 18 and 19. However, here we utilize the `Math` class (instead of `Sum` and `Product`) of the `System` namespace in the .NET Framework class library to find the greater of two numbers using the `Max` method.

```
20:      Console.WriteLine("The maximum number is: " + Math.Max(x, y));
```

The `Math` class contains many useful math-related methods, such as trigonometry and logarithms.

**Tip**

It is permissible to spread one long statement over several lines as long as no tokens are separated into smaller pieces of text. However, if not thoughtfully done, breaking up statements into several lines can easily obscure the readability of the source code. To avoid this problem, choose logical breaking points, such as after an operator or after a comma. When statements are separated into more than one line, indent all successive lines.

For example, we might decide that line 20 of Listing 4.1 is too long

```
20:      Console.WriteLine("The maximum number is: " + Math.Max(x, y));
```

and break it up after the concatenation operator (+) so that `Math.Max(x, y))`; is moved down to the next line and indented

```
20:      Console.WriteLine("The maximum number is: " +
    Math.Max(x, y));
```

Analogous to line 20, `Math.Min(x, y)` in line 21 finds the smaller of `x` and `y`.

```
21:      Console.WriteLine("The minimum number is: " + Math.Min(x, y));
```

This ends our tour through Listing 4.1.

Simplifying Your Code with Methods

We have already discussed how the construction of an object-oriented program can be simplified by breaking it down into suitable classes. The following case study looks at how the design of an individual class can be simplified by breaking its functionality down into suitable methods. It exemplifies and stresses the importance of thinking in terms of building blocks on the method level also.

Methods As Building Blocks: Encapsulating Your Helper Methods with the `private` keyword.

We can expect the internal parts of each class in a program to be less complex than the overall program, but the programmer is still often confronted with individual classes of relatively high complexity. Fortunately, it is possible to reduce this complexity by dividing its functionality into methods. Let's look at an example.

Notice that this discussion is relevant to step 2c (Identification of methods in each class) and step 2d (Internal method design) of the Software Design phase in the Software Development Process presented in Chapter 2.

Consider a map represented in your source code by a class called `Map`. One of the services of this class is to calculate the distance between a list of 6 specified locations (here referred to as `L1`, `L2`, `L3`, `L4`, `L5`, and `L6`) on the map. Every location in the map constitutes a set of 2 coordinates (`x` and `y`), where `L1` has the coordinates (`x1`, `y1`), `L2` has the coordinates (`x2`, `y2`), and so on. The distance between two locations, say `L1` (`x1`, `y1`) and `L2` (`x2`, `y2`), can be calculated by using Pythagoras's formula:

$$\text{distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

which can be used to calculate the distance of a path beginning at `L1` and going through `L2`, `L3` and on to `L6`. The formula for this calculation is

Total distance =

`L1` to `L2` + `L2` to `L3` + `L3` to `L4` + `L4` to `L5` + `L5` to `L6` =

$$\begin{aligned} & \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2} + \\ & \sqrt{(x_3 - x_4)^2 + (y_3 - y_4)^2} + \sqrt{(x_4 - x_5)^2 + (y_4 - y_5)^2} + \sqrt{(x_5 - x_6)^2 + (y_5 - y_6)^2} \end{aligned}$$

We can implement this distance calculation in two ways:

- By using just one method, without any attempt to break up the problem into a couple of simpler methods.
- By thinking in terms of methods as building blocks, we can divide the functionality into several methods.

To keep the examples simple, I will only provide the very important parts of the C# code.

Let's have a closer look at each implementation.

- By using only one method:

```
totalDistance =
    Math.Sqrt(Math.Pow(x1-x2,2) + Math.Pow(y1-y2,2)) +
    Math.Sqrt(Math.Pow(x2-x3,2) + Math.Pow(y2-y3,2)) +
    Math.Sqrt(Math.Pow(x3-x4,2) + Math.Pow(y3-y4,2)) +
    Math.Sqrt(Math.Pow(x4-x5,2) + Math.Pow(y4-y5,2)) +
    Math.Sqrt(Math.Pow(x5-x6,2) + Math.Pow(y5-y6,2));
```

where `Math.Sqrt(x)` calculates the square root of `x` and `Math.Pow(a, b)` raises `a` to the power of `b`.

This coding nightmare is one statement spread over six lines of source code. Let's try to simplify this massive statement.

- Breaking down the functionality into two methods:

Because we are repeatedly calculating distances between two locations, we can separate the functionality into two methods—a helper method calculating distances between two locations and a method calculating the total distance between the six locations. We will simply call our helper method `Distance` and let it contain four formal parameters `a1, b1` (representing the first location) `a2, b2` (representing the second location). When `Distance` is called, it returns the distance between the location arguments sent to it.

We can now write the calculation in only four lines of code.

```
TotalDistance =
    Distance(x1,y1,x2,y2) + Distance(x2,y2,x3,y3) +
    Distance(x3,y3,x4,y4) + Distance(x4,y4,x5,y5) +
    Distance(x5,y5,x6,y6);
```

In this way, we were able to obtain a significant reduction in the complexity of this calculation. Another advantage is that we don't have to remember Pythagoras's formula and try to get it right numerous times; instead, we simply need to call the `Distance` method.



Note

By implementing a class called `Location` to represent specific locations in the form of objects, naming these location objects `L1, L2`, and so on will reduce the previous statement to

```
TotalDistance =
    Distance(L1,L2) + Distance(L2,L3) +
    Distance(L3,L4) + Distance(L4,L5) +
    Distance(L5,L6);
```

Indeed, this is a much simpler and self-documenting statement. To understand and construct this type of statement, we need to put more meat into your understanding for methods and OOP.

If none of our other objects in the program are interested in a `Distance` method, we need to make things less complex when looking at the class from the outside. As a result, we declare this helper method to be `private`.

As we attempt to reduce the complexity of the individual class by breaking its complicated tasks into subtasks, we also create the need for `private` methods.



Tip

A method attempting to solve several tasks is likely to be overly complex and should probably be broken down into smaller, simpler methods. So don't be afraid of creating many small methods.

A method that accomplishes one clear task is said to be *cohesive*.

A good guideline for whether you have created a set of cohesive methods is the ease with which you can name each method. Methods with one distinct task are easier to name than multipurpose methods.

Summary

This chapter presented you with another C# source code example with the ability to perform simple calculations. Many essential constructs and concepts were extracted from this program to extend the knowledge you gained from the previous chapter.

The following are the important elements covered in this chapter:

A variable of type `int` can be used to store whole numbers and can take part in standard arithmetic calculations.

Namespaces help programmers keep their classes organized and accessible to other programmers for reuse purposes.

The `System` namespace in the .NET Framework class library contains many fundamental classes.

Use comments to describe the overall intent of the source code instead of merely restating what the code does.

A `string` value consisting of digits that form a whole number can be converted to a value of type `int`. Conversely, a value of type `int` can also be converted to a value of type `string`.

A method is defined by writing its method header (which includes access modifier, return type, name and formal parameters), by indicating its method body with `{}` and by writing the statements of the method inside the method body.

A method call must include the name of the called method and an argument list that matches the formal parameter list of the called method. When a method is called, the values held by the arguments are assigned to the formal parameters of the called method. A method can return a value, in which case, the method call can be regarded as holding this value just after the method called returns.

When positioned between two numeric values, the `+` operator will perform a standard arithmetic addition but, if positioned between two `strings`, will perform a `string` concatenation.

The `Math` class found in the `System` namespace of the .NET Framework class library contains many helpful methods to perform various mathematical calculations.

Instead of using just one method to solve a complex computational problem, break the problem into several simpler methods. Every method should accomplish just one clear task. Such a method is described as being *cohesive*.

Review Questions

1. What are namespaces used for in C# and .NET?
2. What is the advantage of including the keyword `using` followed by the name of a namespace in the beginning of a program?
3. Which namespace contains classes related to mathematical calculations and console input/output.
4. How should comments be applied in the source code?
5. Describe a variable of type `int`.
6. Why are `x` and `y` often regarded as unacceptable variable identifiers? Why are they acceptable in Listing 4.1?
7. What are the fundamental parts of a method?
8. Why is `MoveLeft` a bad name for a class? For which C# construct would it be better suited?
9. How do you specify that a method does not return a value?
10. How do you specify that a method returns a value of type `int`?
11. What are arguments (in method calls)?
12. What are formal parameters?
13. How do arguments and formal parameters relate?
14. Does the `+` operator only perform arithmetic additions?
15. How can you break down the inner complexities of a class?
16. What is a cohesive method?

Programming Exercises

Make the following changes to the program in Listing 4.1:

1. Change the multi-line comments in lines 3–6 to two single line comments.
2. Apart from addition and multiplication, allow the user to perform a subtraction. Among other changes, you need to add a **Subtract** method. (The symbol – is used to perform subtractions in C#).
3. Instead of calculating the sum and the product of *two* numbers, make the program perform the calculations on *three* numbers. (You can ignore the **Max** and **Min** functions here.) Hint: You need to declare another **int** variable in **Main**. The **Sum**, **Product**, and **Subtract** methods must accept three arguments instead of two. You must allow the user to input a third number, and you must include the third argument when these methods are called.
4. Create two methods called **MyMax** and **MyMin** that both take three arguments and find the maximum and minimum value of these arguments. Hint: **Math.Max(Math.Max(a, b), c)** returns the max of **a**, **b** and **c**.

CHAPTER 5

YOUR FIRST OBJECT-ORIENTED C# PROGRAM

You will learn about the following in this chapter:

- The atomic elements of a C# source program
- The conventional styles used for naming classes, methods, and variables
- Operators, operands, and expressions (introductory)
- How to write and instantiate your own custom-made classes
- How the theoretical OO discussion about Elevators and Person classes in Chapter 3, “A Guided Tour Through C#: Part I,” can be implemented to form a fully working C# program
- What a simple object-oriented program looks like and its important elements
- How to initialize the instance variables of your newly created objects
- The ability of objects to contain instance variables that contain other objects
- How programmers implement relationships between classes to let them collaborate and form object-oriented programs
- The Unified Modeling Language (UML) and how it can be used to graphically illustrate and model relationships among classes
- Three common types of relationships among classes called association, aggregation, and composition

Introduction

You have now been presented with two source programs—Listing 3.1 of Chapter 3 and Listing 4.1 of Chapter 4, “A Guided Tour through C#: Part II.” The associated presentation of the C# language constructs has been somewhat guided by the contents of each particular line of code and, consequently, touched on many diverse but interrelated aspects simultaneously. To make up for this fast tour through C#, the first part of this chapter provides an overview of the very basic elements of C#.

The last part of the chapter contains the first object-oriented C# program of the book. It builds on previous theoretical OO discussions, in particular the `Elevator` simulation discussion in the beginning of Chapter 3. Among other things, it allows you to see how the relationship between the `Elevator` and the `Person` classes discussed in this earlier section is implemented in C#.

Lexical Structure

When the C# compiler receives a piece of source code to compile, it is faced with the seemingly daunting task of deciphering a long list of characters (more specifically Unicode characters, presented in Appendix E, “Unicode Character Set”) which can be found at www.samspublishing.com and turn them into matching MSIL with exactly the same meaning as the original source code. To make sense of this mass of source code, it must recognize the atomic elements of C#—the unbreakable pieces making up the C# source code. Examples of atomic elements are a brace {}, a parenthesis (), and keywords like `class` and `if`. The task performed by the compiler, associated with differentiating opening and closing braces, keywords, parentheses, and so on is called *lexical analysis*. Essentially, the lexical issues dealt with by the compiler pertain to how source code characters can be translated into tokens that are comprehensible to the compiler.

C# programs are a collection of identifiers, keywords, whitespace, comments, literals, operators, and separators. You have already met many of these C# elements. The following provides a structured overview of these and, whenever relevant, will introduce a few more aspects.

Identifiers and CaPiTaLiZaTioN Styles

Identifiers are used to name classes, methods, and variables. We have already looked at the rules any identifier must follow to be valid and how well-chosen identifiers can enhance the clarity of source code and make it self-documenting. Now we will introduce another aspect related to identifiers—namely CaPiTaLiZaTioN sTyLe.

Often programmers choose identifiers that are made up of several words to increase the clarity and the self-documentation of the source code. For example, the words could be child-births-per-year. However, because of the compiler’s sensitivity to whitespace, any identifier broken up into words by means of whitespace will be misinterpreted. For example, a variable to represent the average speed per hour cannot be named `average speed per hour`. We need to discard whitespace to form one proper token, while maintaining a style allowing the reader of the source code to distinguish the individual words in the identifier. Some computer languages have agreed on the convention `average_speed_per_hour`. In C#, however, most programmers utilizes an agreed-upon sequence of upper- and lowercase characters to distinguish between individual words in an identifier.

A couple of important capitalization styles are applied in the C# world:

- *Pascal casing*—The first letter of each word in the name is capitalized, as in AverageSpeedPerHour.
- *Camel casing*—Same as Pascal casing, with the exception of the first word of the identifier that is lowercase, as in averageSpeedPerHour.

Pascal casing is recommended when naming classes and methods, whereas Camel casing is used for variables.



Tip

Not all computer languages are case sensitive. In these languages, AVERAGE and average are identical to the compiler. For compatibility with these languages, you should avoid using case as the distinguishing factor between `public` identifiers accessible from other languages.

Literals

Consider the following two lines of source code:

```
int number;
number = 10;
```

`number` is clearly a variable. In the first line, we declare `number` to be of type `int`. In the second line, we assign `10` to `number`. But what is `10`? Well, `10` is incapable of changing its value and is named a *literal*. Literals are not confined to numbers. They can also be characters, such as `B`, `$`, and `z` or text, such as “This is a literal.” Literals can be stored by any variable with a type compatible with that of the literal.

Comments and Source Code Documentation

The main characteristic of comments is the compiler’s ability to totally ignore them. We have so far seen two ways of making comments—single line with `//` and multi-line using `/* */`.

In fact, there is a third type that allows you to write the documentation as part of the source code as shown in this chapter, but with the added ability of extracting this documentation into separate Extensible Markup Language (XML) files. For now, you can appreciate a particular useful end result of this feature; you just need to take a look at the .NET Framework class library documentation, which was created by extracting XML files from the comments/documentation sitting inside the original source code.

Separators

Separators are used to separate various elements in C# from each other. You have already met many of them. An example is the commonly used semicolon `;` that is required to terminate a statement. Table 5.1 summarizes the separators we have presented to so far.

TABLE 5.1 Important Separators in C#

Name	Symbol	Purpose
Braces	{ }	Used to confine a block of code for classes, methods, and the, yet to be presented, branching and looping statements.
Parentheses	()	Contains lists of formal parameters in method headers and lists of arguments in method invocation statements. Also required to contain the Boolean expression of an if statement and other, yet to be presented, branching and looping statements.
Semicolon	;	Terminates a statement.
Comma	,	Separates formal parameters inside the parentheses of a method header and separates arguments in a method invocation statement.
Period (dot operator)	.	Used to reference namespaces contained inside other namespaces and to specify classes inside namespaces and methods (if accessible) inside classes and objects. It can also be used to specify instance variables inside classes and objects (if accessible), but this practice should be avoided.

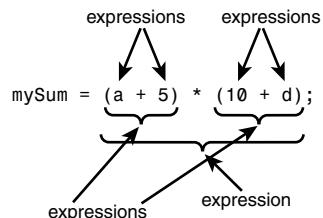
Operators

Operators are represented by symbols such as +, =, ==, and *. Operators act on *operands*, which are found next to the operator. For example

`SumTotal + 10`

contains the + operator surrounded by the two operands `sumTotal` and `10`. In this context, the + operator combines two operands to produce a result and so it is classified as a *binary operator*. Some operators act on only one operand; they are termed *unary operators*.

Operators, together with their operands, form *expressions*. A literal or a variable by itself is also an expression, as are combinations of literals and variables with operators. Consequently, expressions can be used as operands as long as the rules, which apply to each individual operator, are adhered to, as shown in the following example:



`a`, `5`, and `10`, `d` are all expressions acted on by the + operator. However, `(a + 5)` and `(10 + d)` are also expressions acted on by the * operator. Finally, `(a + 5) * (10 + d)` can be regarded

as one expression. The assignment operator = acts on this latter expression and the expression `mySum`. Expressions are often nested inside each other to form hierarchies of expressions, as in the previous example.

Operators can be divided into the following categories—assignment, arithmetic, unary, equality, relational, logical, conditional, shift, and primary operators.

We will spend more time on operators in later chapters, but the following is a quick summary of the operators you have encountered so far:

- *Assignment operator* (=)—Causes the operand on its left side to have its value changed to the expression on the right side of the assignment operator as in

```
29:         sumTotal = a + b;
```

where `a + b` can be regarded as being one operand.

- Binary *arithmetic operators* (+ and *)—The following example

```
a * b
```

multiplies `a` and `b` without changing their values.

- *Concatenation operator* (+)—Concatenates two strings into one string.

- *Equality operator* (==)—Compares two expressions to test whether they are equal. For example,

```
leftExpression == rightExpression
```

will only be true if the two expressions are equal; otherwise, it is false.

Keywords

Appendix C at www.samspublishing.com lists all 77 different keywords of C#. We have so far met the keywords `if`, `class`, `public`, `static`, `void`, `string`, `int`, and `return`. The syntax (language rules) of the operators and separators combined with the keywords form the definition of the C# language.

Some Thoughts on Elevator Simulations

The C# program presented after the following case study is a first attempt (prototype) to implement an object-oriented elevator simulation. The case study introduces a few goals, problems, and solutions of a simple elevator simulation and attempts to put you in the right frame of mind for the following practical C# example. It is also meant as a reminder of our encapsulation discussion in the beginning of Chapter 3.

Concepts, Goals and Solutions in an Elevator Simulation Program: Collecting Valuable Statistics for Evaluating an Elevator System

This case study looks at a strategy for collecting important information from a particular elevator simulation to answer the question, “Is the elevator system that we are simulating performing its task of transporting passengers between various floors properly?” In other words, if we took a real elevator system, extracted all its relevant characteristics, and turned it into a computer simulation, how well would this system perform?



Statistics

A numerical fact or datum, especially one computed from a sample, is called a *statistic*.

Statistics is the science that deals with the collection, classification, analysis, and interpretation of numerical facts or data, and that, by use of mathematical theories of probability, imposes order and regularity on aggregates of more or less disparate elements.

The term statistics is also used to refer to the numerical facts or data themselves.

Two statistics considered important by most elevator travelers when rating an elevator system are

- The time a typical person has to wait on a floor after having pressed the button to call the elevator
- The average time it takes to travel one floor

An attempt to collect these numbers from an elevator simulation could be done as follows.

Because the `Person` objects are “traveling” through the elevator system, it is ultimately these objects that the `Elevator` objects must service. Consequently, the `Person` objects will “know” how well the system is working and are able to collect some of the essential statistics required. An analogous strategy in the real world would be to interview the users of the elevator system and gather some of their elevator system experiences.

Each `Person` object of the elevator simulation program could be implemented to have a couple of instance variables keeping track of its total waiting time outside the elevator (to answer the first bullet) and an average traveling time per floor (addressing the second bullet). They would give a good indication of how well the elevator system is working and be part of the statistics collected for each simulation. We could call these variables `totalWaitingTime` and `averageFloorTravelingTime`.

Calculating `totalWaitingTime` would require a method located inside `Person` containing instructions to start the computer’s built-in stopwatch every time the person has “pressed a button” on a particular `Floor` object calling an elevator. As soon as the `Elevator` object “arrives,” the stopwatch is stopped and the time is added to the current value of `totalWaitingTime`.

Similarly, the `averageFloorTravelingTime` is calculated by another method inside `Person` starting the stopwatch as soon as the `Person` object has “entered” the `Elevator` object. When the `Person` has reached its “destination,” the stopwatch is stopped and the time divided by the number of floors “traveled” to get the average traveling time per floor. This result is stored in a list together with other `averageFloorTravelingTime` sizes. When the final `averageFloorTravelingTime` statistic needs to be calculated, a method will calculate the average of the numbers stored in the list.

All these calculations involving starting and stopping stopwatches, summing numbers, calculating averages, and so on are not of any interest to other objects in the simulation, and they would complicate matters unduly for other programmers were they exposed to them.

Consequently, we should hide all the methods involved here by declaring them to be `private`.

Each `Person` object must also be able to report its `totalWaitingTime` and `averageFloorTravelingTime` sizes. We can do this through two `public` methods arbitrarily named `getTotalWaitingTime` and `getAverageFloorTravelingTime`. Any other object calling any of these two methods will receive the corresponding statistic.

Another programmer who is also working on this project is writing a class to collect the important statistics of each simulation. He or she calls this class `StatisticsReporter`. He or she must ensure that all `Person` objects are “interviewed” at the end of a simulation by letting the `StatisticsReporter` collect their `totalWaitingTime` and `averageFloorTravelingTime` statistics. The `StatisticsReporter` can now simply do this by calling the `getTotalWaitingTime` and `getAverageFloorTravelingTime` methods of each `Person` object involved in a particular simulation.

To summarize:

- `getTotalWaitingTime` and `getAverageFloorTravelingTime` are part of an interface that is hiding or encapsulating all the irrelevant complexities for our happy programmer of the `StatisticsReporter`.
- Likewise, the instance variables of the `Person` objects should be hidden by declaring them `private`. This prevents any other objects, including the `StatisticsReporter`, from mistakenly changing any of these sizes. In other words, the `getTotalWaitingTime` and `getAverageFloorTravelingTime` methods “cover,” by means of encapsulation, the instance variables `totalWaitingTime` and `averageFloorTravelingTime` as a protective wrapper by allowing only the `StatisticsReporter` to get the values of these and not to set them.

Object-Oriented Programming: A Practical Example

So far, our discussion about object-oriented programming has been somewhat theoretical. We have discussed the difference between classes and objects and how objects need to be instantiated before they can actually perform any actions. You have seen a couple of simple classes in Listings 3.1 of Chapter 3 and 4.1 of Chapter 4, but they didn’t contain any real objects; they

were passive containers only created to hold the `Main` method. Actually, none of these programs can be regarded as being particularly object-oriented. Because they only have methods containing sequences of statements one executed after the other, they resemble programs written in the procedural programming style.

By merely adding plenty of methods and statements to the `SimpleCalculator` class of Listing 4.1, it would be possible, albeit extremely cumbersome, to write a valid and full-blown sophisticated spreadsheet application without ever being concerned about writing any other classes, learning any theoretical object-oriented principles, or ever using any mentionable object-oriented C# features. The structure of this program would be closer to a program written in a procedural language like C or Pascal than it would to a typical object-oriented C# program.

Conversely and surprisingly, it would also be possible to write an object-oriented program in C or Pascal, but it would be awkward because these languages, as opposed to C#, do not have any built-in support for this paradigm.

To make our previous object-oriented discussions more practical and to avoid the risk of constructing a non-object-oriented full-size spreadsheet program, I have provided an example illustrating a couple important C# features closely related to our theoretical discussion about classes, objects, and instantiation.

Presenting SimpleElevatorSimulation.cs

Listing 5.1 contains the source code for a simple elevator simulation program. Its goal is to illustrate what a custom-made object looks like in C# and how it is instantiated from a custom written class. In particular, it shows how an `Elevator` object is created and how it calls a method of the `Person` object named `NewFloorRequest` and triggers the latter to return the number of the requested “floor,” enabling the `Elevator` to fulfill this request.

Many abstractions were made during the design stage of the source code in Listing 5.1, allowing us to make the program simple and be able to concentrate on the essential object-oriented parts of the source code.

The following is a brief description of the overall configuration of the elevator’s system used for this simulation, highlighting the major differences to a real elevator system.

- The `Building` class has one `Elevator` object called `elevatorA`.
- One `Person` object residing inside the `passenger` variable is “using” `elevatorA`.
- The `Elevator` object can “travel” to any “floor” that is within the range specified by the `int` type (-2147483648 to 2147483647). However, a `Person` object is programmed to randomly choose floors between 1 and 30.
- The elevator will instantly “travel” to the “destination floor.”
- The “movements” of `elevatorA` will be displayed on the console.
- After the `passenger` has “entered” `elevatorA`, “he” or “she” will stay in `elevatorA` throughout the simulation and simply choose a new floor whenever a previous request has been satisfied.

- Only the `Elevator`, `Person`, and `Building` classes are used in this simulation, leaving out `Floor`, `StatisticsReporter`, and other classes considered important for a full-blown simulation.
- At the end of each simulation, the total number of floors traveled by `elevatorA` will be displayed on the console. This number could be a very important statistic in a serious elevator simulation.

So, despite the simplicity and high abstraction of this simulation, we are actually able to extract one important statistic from it and, without too many additions, you would be able to create a small but useful simulation enabling the user to gain valuable insights into a real elevator system.

Please take a moment to examine the source code in Listing 5.1. Try first to establish a bigger picture when looking through the code. For example, notice the three class definitions (`Elevator`, `Person` and `Building`) that constitute the entire program (apart from lines 1–4). Then notice the methods defined in each of these three classes and the instance variables of each class.



Note

Recall that the order in which the methods of a class are written in the source code is independent of how the program is executed. The same is true for the order of the classes in your program. You can choose any order that suits your style. In Listing 5.1, I chose to put the class containing the `Main` method last, and yet `Main` is the first method to be executed.

Typical output from Listing 5.1 is shown following the listing. Because the requested floor numbers are randomly generated, the "Departing floor" and "Traveling to" floors (except for the first departing floor, which will always be 1) and the "Total floors traveled" will be different on each run of the program.

LISTING 5.1 Source Code for `SimpleElevatorSimulation.cs`

```

01: // A simple elevator simulation
02:
03: using System;
04:
05: class Elevator
06: {
07:     private int currentFloor = 1;
08:     private int requestedFloor = 0;
09:     private int totalFloorsTraveled = 0;
10:     private Person passenger;
11:
12:     public void LoadPassenger()
13:     {
14:         passenger = new Person();
15:     }

```

LISTING 5.1 continued

```
16: 
17:     public void InitiateNewFloorRequest()
18:     {
19:         requestedFloor = passenger.NewFloorRequest();
20:         Console.WriteLine("Departing floor: " + currentFloor
21:                           + " Traveling to floor: " + requestedFloor);
22:         totalFloorsTraveled = totalFloorsTraveled +
23:             Math.Abs(currentFloor - requestedFloor);
24:         currentFloor = requestedFloor;
25:     }
26: 
27:     public void ReportStatistic()
28:     {
29:         Console.WriteLine("Total floors traveled: " + totalFloorsTraveled);
30:     }
31: }
32: 
33: class Person
34: {
35:     private System.Random randomNumberGenerator;
36: 
37:     public Person()
38:     {
39:         randomNumberGenerator = new System.Random();
40:     }
41: 
42:     public int NewFloorRequest()
43:     {
44:         // Return randomly generated number
45:         return randomNumberGenerator.Next(1,30);
46:     }
47: }
48: 
49: class Building
50: {
51:     private static Elevator elevatorA;
52: 
53:     public static void Main()
54:     {
55:         elevatorA = new Elevator();
56:         elevatorA.LoadPassenger();
57:         elevatorA.InitiateNewFloorRequest();
58:         elevatorA.InitiateNewFloorRequest();
59:         elevatorA.InitiateNewFloorRequest();
60:         elevatorA.InitiateNewFloorRequest();
61:         elevatorA.InitiateNewFloorRequest();
62:         elevatorA.ReportStatistic();
63:     }
64: }
```

Departing floor: 1 Traveling to floor: 2
Departing floor: 2 Traveling to floor: 24

```
Departing floor: 24 Traveling to floor: 15
Departing floor: 15 Traveling to floor: 10
Departing floor: 10 Traveling to floor: 21
Total floors traveled: 48
```

Overall Structure of the Program

Before we continue with the more detailed analysis of the program, look at Figure 5.1. It connects the illustration used in Figure 3.1, shown in Chapter 3, with the concrete C# program of Listing 5.1.

The `Elevator` and `Person` classes of Listing 5.1 define abstracted versions of our now familiar counterparts from the real world. They are graphically depicted next to their C# counterparts in Figure 5.1. Each part of the `Elevator` and `Person` classes written in the C# source code (indicated by braces) has been linked to its graphical counterpart with arrows. Notice how the `public` methods of the two classes (the interface) encapsulate the hidden `private` instance variables. In this case, no `private` methods were needed.

The `Building` class has one `Elevator`, which is represented by its `elevatorA` instance variable declared in line 51. It also holds the `Main` method where the application commences. This class is never instantiated in the program. It is used by the .NET runtime to access `Main` and start the program.

Just as in the previous listings, I will provide a brief explanation of the lines in the source code. Many lines have not been shown because they have already been explained in a previous example.

LISTING 5.2 Brief Analysis of Listing 5.1

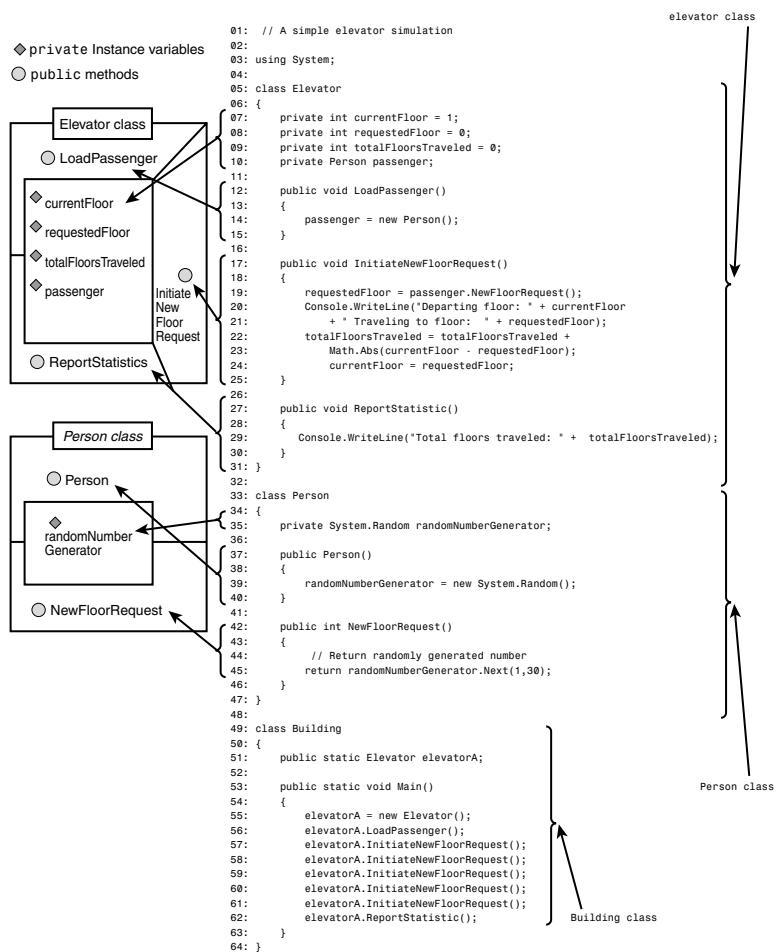
- 05: Begin the definition of a class named `Elevator`
- 07: Declare an instance variable called `currentFloor` to be of type `int`; set
 - its access level to `private` and its initial value to 1.
- 10: Declare `passenger` to be a variable, which can hold an
 - object of class `Person`. The class `Person` is said to play the role
 - of `passenger` in its association with the `Elevator` class.
- 12: Begin the definition of a method called `LoadPassenger`. Declare
 - it `public` to be part of the interface of the `Elevator` class.
- 14: Instantiate (create) a new object of class `Person` by using the
 - `new` keyword. Assign this object to the `passenger` variable.
- 17: Begin the definition of a method called `InitiateNewFloorRequest`. Declare
 - it `public` to be part of the interface of the `Elevator` class.
- 19: Call the `NewFloorRequest` method of the `passenger` object; assign the
 - number returned by this method to the `requestedFloor` variable.

LISTING 5.2 continued

- 20-21: Print information about the "movement" of the operator
↳on the command console.
 - 22-23: Calculate the number of floors traveled by the elevator on one
↳particular trip (line 23). Add this result to the total number
↳of floors already traveled by the elevator.
 - 24: Let the Elevator "travel" to the requested floor by assigning the
↳value of requestedFloor to currentFloor.
 - 29: Whenever the ReportStatistics method is called, print out the
↳totalFloorsTraveled variable.
 - 33: Begin the definition of a class named Person.
 - 35: Declare randomNumberGenerator to be a variable, which can hold an
↳object of class System.Random.
 - 37: Begin the definition of a special method called a constructor, which will
↳be invoked automatically whenever a new object of class
↳Person is created.
 - 39: Create a new object of class System.Random by using the
↳C# keyword: new; assign this object to the
↳randomNumberGenerator variable.
 - 42: Begin the definition of a method called NewFloorRequest. public declares
↳it to be part of the interface of the Person class. int specifies
↳it to return a value of type int.
 - 43: The Person decides on which floor "he" or "she" wishes to travel to
↳by returning a randomly created number between 1 and 30.
 - 51: The Building class declares an instance variable of type Elevator
↳called elevatorA. The Building class is said to have a "has-a"
↳(or a composition) relationship with the Elevator class.
 - 53: Begin the definition of the Main method where the program will commence.
-
- 55: Instantiate an object of class Elevator with the keyword new; assign
↳this object to the elevatorA variable.
 - 56: Invoke the LoadPassenger method of the elevatorA object.
 - 57-61: Invoke the InitiateNewFloorRequest method of the
↳elevatorA object 5 times.
 - 62: Invoke the ReportStatistic method of the elevatorA object.

FIGURE 5.1

Linking Figure 3.1 with actual C# program.



A Deeper Analysis of `SimpleElevatorSimulation.cs`

The following sections discuss important subjects related to Listing 5.1.

Defining a Class to Be Instantiated in Our Own Program

You are already familiar with how a class is defined. I still highlight line 5 because this is the first time you define your own class that is being instantiated and therefore used to create an object in your program.

`05: class Elevator`

Initializing Variables

The new feature presented in line 7 is the combination of a declaration statement and an assignment statement. This is called an *initialization*. `private int currentFloor;` is a

straightforward declaration and, by adding = 1 at the end, you effectively assign 1 to `currentFloor` during or immediately after the creation of the object to which this instance variable belongs.

```
07:     private int currentFloor = 1;
```

A variable often needs an initial value before it can be utilized in the computations in which it takes part. `currentFloor` in line 7 is a good example. We want the elevator to start at floor number 1 so we initialize it to this value.

When a variable is assigned a value before it takes part in any computations, we say it is being *initialized*. Two important methods exist to initialize an instance variable. You can

- Assign the variable an initial value in its declaration (as in line 7).
- Utilize a C# feature called a constructor. The source code contained in a *constructor* is executed when an object of a class is being created, which is the ideal time for any initializations. Constructors are informally presented shortly.

Instance variables that are not explicitly initialized in the source code are automatically assigned a default value by the C# compiler. For example, had we not assigned the value 1 to `currentFloor` in line 7, C# would have given it the default value 0.



Initialize All Your Variables Explicitly

Do not rely on the C# compiler to initialize your variables. Explicit initializations make the code clearer and avoid the source code relying on compiler initializations and their default values, which can change in the future and introduce errors into your source code.



Tip

It is possible to declare a class member without explicitly stating its accessibility by leaving out the `public` or `private` keyword. The accessibility is then, by default, `private`.

Nevertheless, use the `private` keyword to declare all `private` class members to enhance clarity.

Declaring a Variable Representing an Object of a Specific Class

Line 10 states that the instance variable `passenger` can hold an object created from the `Person` class.

```
10:     private Person passenger;
```

We haven't yet put a particular `Person` object here; we would need an assignment statement to do this, which you will see a bit further in this section. So far, we are merely expressing that an `Elevator` object is able to "transport" one `passenger`, which must be a `Person`. For example, no `Dog`, `Airplane`, or `Submarine` objects can be stored in the `Elevator`, had we ever defined such classes in our program.

Now, jump down to lines 33–47 for a moment. By defining the `Person` class in these lines, you have effectively created a new custom-made type. So, instead of merely being able to declare a variable to be of type `int` or type `string`, you can also declare it to be of type `Person`, which is exactly what you do in line 10.



Note

`int` and `string` are built-in, pre-defined types. The classes you write and define in your source code are custom-made types.

Notice that line 10 is closely linked with lines 14, 19, and 33–47. Line 14 assigns a new `Person` object to `passenger`; line 19 utilizes some of the functionality of the `passenger` variable (and hence a `Person` object) by calling one of its methods, and lines 33–47 define the `Person` class.



Tip

The sequence of class member declarations and definitions is arbitrary. However, try to divide the class members into sections containing members with similar access modifiers to improve clarity.

The following is one example of a commonly used style:

```
class ClassName
{
    declarations of private instance variables
    private method definitions
    public method definitions
}
```

Instance Variables Representing the State of an Elevator Object

Lines 7–10 contain the list of instance variables I found relevant to describe an `Elevator` object for this simulation.

```
07:     private int currentFloor = 1;
08:     private int requestedFloor = 0;
09:     private int totalFloorsTraveled = 0;
10:     private Person passenger;
```

The state of any `Elevator` object is described by the instance variables declared in the following lines:

- *Line 7*—The `currentFloor` variable keeps track of the floor on which an `Elevator` object is situated.
- *Line 8*—A new floor request will be stored in the `requestedFloor` variable. The `Elevator` object will attempt to fulfill this request as swiftly as possible (line 24), depending on the speed of your computer's processor.

- **Line 9**—When the `Elevator` object is created, you can regard it to be “brand new.” It has never moved up or down, so `totalFloorsTraveled` initially must contain the value 0. This is achieved by initializing it to the value 0.

The amount of floors traveled is added to `totalFloorsTraveled` (lines 22–23), just before a trip is finished (line 24).

- **Line 10**—The passenger(s) of the elevator are ultimately the decision makers of the floor numbers visited by the elevator. The `Person` object residing inside `passenger` chooses which floors our `Elevator` object must visit. A request is obtained from the `passenger` in line 19 and assigned to the `requestedFloor` variable.



Abstraction and the Choice of Instance Variables

Recall the discussion about abstraction in the beginning of Chapter 3. The goal associated with abstraction is to identify the essential characteristics of a class of objects that are relevant to our computer program.

During the process of deciding which instance variables to include in a class definition, and the declaration of them in the source code, the programmer is applying the conceptual idea of abstraction in a very practical, hands-on manner.

For example, I could have attempted to include an instance variable such as `color` of type `string` in the `Elevator` class, declared as follows:

```
private string color; ← probably useless for our purposes
```

But where can I make any use of it? I could perhaps assign the `string red` to it and write a method that would print the following:

```
My color is: red
```

on the command console whenever called; but the exercise would be irrelevant to what we are trying to achieve in our simple little simulation so it is wasteful and complicates our `Elevator` class unnecessarily.

Another programmer might have included an instance variable of the `Elevator` class that measures the number of trips performed by the elevator; the programmer might call it `totalTrips` and declare it as follows:

```
private int totalTrips; ← potentially useful
```

The `Elevator` class could then be designed so that a method would add 1 to `totalTrips` every time a request had been fulfilled. This instance variable would enable us to keep track of another, perhaps important, statistic and so it has the potential of being useful.

As you can see, there are many different ways to represent a real world object when deciding which instance variables to include. The choice of instance variables depends on the individual programmer and what he or she wants to do with each object.

Enabling Elevator Object to Load a New Passenger

An `Elevator` object must be able to load a new `Person` object. This is accomplished by the `LoadPassenger` method residing inside an `Elevator` object (see line 12). `LoadPassenger()` is accessed from outside its object, and so it must be declared `public`. The call to `LoadPassenger` is made in line 56 of Listing 5.1.

```
12:    public void LoadPassenger()
```



Talking About Classes and Objects

Consider the following declaration statement:

```
private Person passenger;
```

You will come across many different, more or less correct, ways to describe what this sentence is about and, in particular, how to identify an object. In my opinion, the following is a good description but perhaps a bit cumbersome:

"`passenger` is a variable declared to hold an object of class `Person`."

Due to this lengthy description, I often use the following expression:

"`passenger` is a variable declared to hold a `Person` object."

In Chapter 6, "Types Part I: The Simple Types," you will see why the following description is probably the most correct but also the longest:

"`passenger` is a variable declared to hold a reference to an object of class `Person`."

Creating a New Object with the `new` Keyword

Line 14 goes hand-in-hand with line 10.

```
14:    passenger = new Person();
```

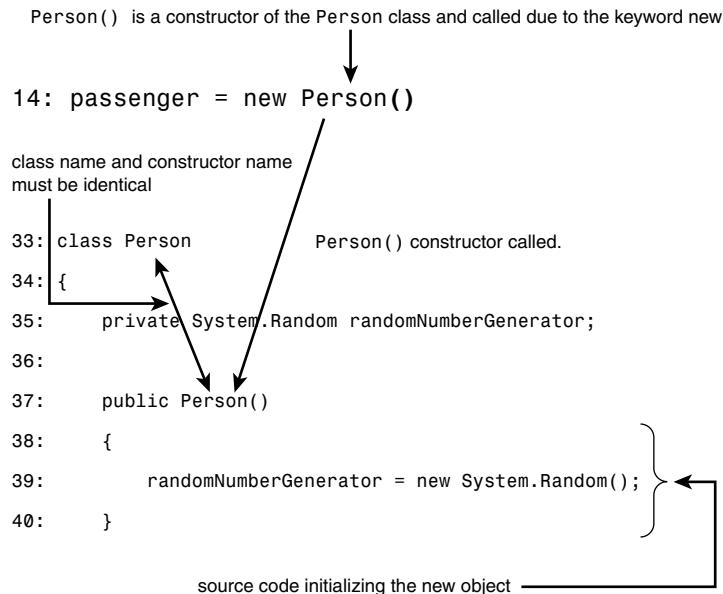
`new`, as applied in this line, is a keyword in C# used to instantiate a new object. It will create a new instance (object) of the `Person` class. This new object of class `Person` is then assigned to the `passenger` variable. `passenger` now contains a `Person` object that can be called and used to perform actions.



Note

When an object is instantiated, an initialization of this object's instance variables is often needed. A *constructor* is a special kind of method that performs this initialization task.

To specify that a method is a constructor, it must have the same name as its class. Thus, a constructor for the `Person` class is called `Person()` (see line 37). Whenever a new `Person` object is created with the keyword `new`, the `Person()` constructor is automatically called to perform the necessary initializations. This explains the parentheses in line 14, which are always required when any method, and any constructor, is called.



Letting an Elevator Object Receive and Fulfill Passenger Requests

By calling the `InitiateNewFloorRequest` method, the `Elevator` object is briefly instructed to:

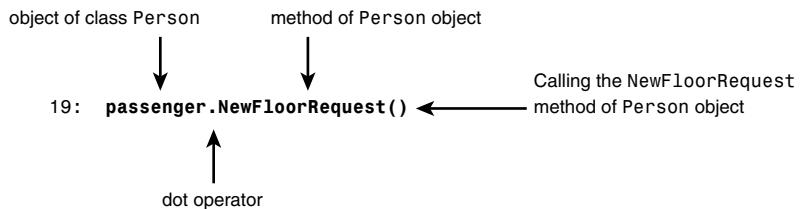
- Get a new request from the `passenger` (line 19)
- Print out its departure and destination floor (lines 20-21)
- Update the `totalFloorsTraveled` statistic (lines 21-22)
- Fulfill the request of the `passenger` (line 24).

```

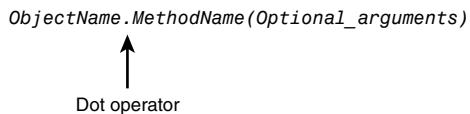
17:     public void InitiateNewFloorRequest()
18:     {
19:         requestedFloor = passenger.NewFloorRequest();
20:         Console.WriteLine("Departing floor: " + currentFloor
21:                           + " Traveling to floor: " + requestedFloor);
22:         totalFloorsTraveled = totalFloorsTraveled +
23:             Math.Abs(currentFloor - requestedFloor);
24:         currentFloor = requestedFloor;
25:     }
  
```

Let's begin with line 19:

In line 19 the `NewFloorRequest` method of `passenger` is called with:



Here, we are using the following syntax introduced earlier:



where the dot operator (`.`) is used to refer to a method residing inside an object. You have already used the dot operator many times. For example you used the dot operator when you called the `WriteLine` method of the `System.Console` with `System.Console.WriteLine("Bye Bye!")`. This time, however, instead of calling a prewritten method of the .NET Framework you are calling your own custom-made method from the `Person` class.



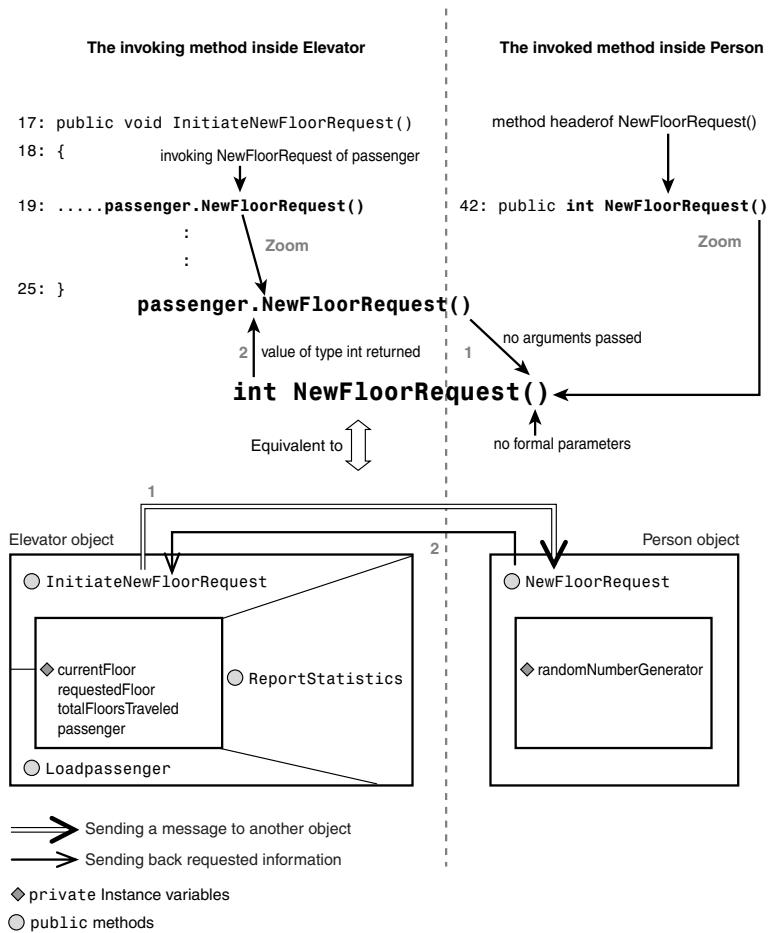
Note

You can see the fixed class definition in the source code. An object, on the other hand, is dynamic and comes to life during execution of the program.

Instead of calling a method residing inside the same object, you are calling a method residing in another object. Here, the `Elevator` object is said to send a message to the `Person` object. A call to a method residing in another object is similar to calling a local method. In Figure 5.2, I have linked Figure 3.1 with the general mechanism illustrated in Figure 4.1. The upper half of the figure illustrates the call to `NewFloorRequest`, following the same graphical style as in Figure 4.1. Step 1 symbolizes the call to `NewFloorRequest`. There are no formal parameters specified for `NewFloorRequest`, so no arguments are passed to the method. This step is equivalent to step 1 of the lower half of the figure. After the `NewFloorRequest` has terminated, it returns a value of type `int`, as symbolized by the arrow marked 2. The graphics equivalent of step 2 is shown in the lower half of this figure. As seen previously (line 18 of Listing 4.1 in Chapter 4), after step 2 is completed, you can substitute `passenger.NewFloorRequest()` with this value of type `int`. Finally, this value can be assigned to the variable `requestedFloor` with the assignment operator `=`.

FIGURE 5.2

Invoking a method of another object.



After the newly received request from its `passenger` in line 19, the `Elevator` object has a current position held by the `currentFloor` variable and a different floor held in `requestedFloor` to which it must travel. If the `Elevator` is “brand new” and has just been created, `currentFloor` will have a value of 1. If the `Elevator` has already been on one or more “rides,” `currentFloor` will be equal to the destination of the latest “ride.” Line 20 will print out the value of the `currentFloor` by using the now familiar `WriteLine` method. The newly received `passenger` request is then printed in line 21.

Note that lines 22 and 23 represent just one statement; you can verify this by locating the single semicolon found at the end of line 23. The statement has been spread over two lines due to lack of space. Line 23 has been indented relative to line 22; this indicates to the reader that we are dealing with only one statement.



Absolute Value

The *absolute value* of a positive number is the number itself.

The absolute value of a negative number is the number with the negative sign removed.

For example,

The absolute value of (-12) is 12.

The absolute value of 12 is 12.

In Mathematics, absolute value is indicated with two vertical lines, surrounding the literal or variable, as shown in the following:

$$|-12| = 12$$



The use of *Math.Abs()*

The *Abs* method of the *Math* class residing inside the .NET Framework returns the absolute value of the argument sent to it. Consequently, the method call *Math.Abs(99)* returns 99, whereas *Math.Abs(-34)* returns 34.

When calculating the distance traveled by the elevator, we are interested in the positive number of floors traveled, irrespective of whether the elevator is moving up or down. If we calculate the number of floors traveled using the following expression:

```
currentFloor - requestedFloor
```

we will end up with a negative number of floors traveled whenever *requestedFloor* is larger than *currentFloor*. This negative number is then added to *totalFloorsTraveled*; so the number of floors traveled in this case has mistakenly been deducted rather than added from *totalFloorsTraveled*. We can avoid this problem by adding the absolute value of (*currentFloor* - *requestedFloor*), as has been done in line 23:

```
Math.Abs(currentFloor - requestedFloor);
```

The statement can be broken down into the following sub-instructions:

1. *Math.Abs(currentFloor - requestedFloor)*—Calculate the number of floors traveled on the next elevator ride.
2. *totalFloorsTraveled + Math.Abs(currentFloor - requestedFloor)*—Add the number of floors traveled on the next elevator ride to the current value of *totalFloorsTraveled*.
3. Assign the result of 2 to *totalFloorsTraveled*.

Having *totalFloorsTraveled* on both sides of the assignment operator might seem odd at first. However, this is a very useful operation adding the number of floors traveled on the latest elevator ride to the original value of *totalFloorsTraveled*. *totalFloorsTraveled* is the “odometer” of the elevator, constantly keeping track of the total amount of floors it has traveled. More details about this type of statement will be provided in Chapter 6.

Surprisingly, the simple assignment statement in line 24 represents the “elevator ride”. It satisfies the request from *passenger*. By assigning the value of the *requestedFloor* variable to

`currentFloor`, we can say that the elevator has “moved” from the `currentFloor` value it contained prior to this assignment to the `requestedFloor`.

The Person Class: The Template for an Elevator Passenger

Line 33 begins the definition of the second custom written class, which will be used as a template to create an object.

```
33: class Person
```

As we have seen, a `Person` object is created and kept inside an `Elevator` object from which its `NewFloorRequest` method is called, telling the `Elevator` object where its `passenger` wants to go next.

In Line 35, an object of the `Person` class has been enabled to make floor requests by equipping it with an object containing a method that can generate random numbers. This object is kept in the `randomNumberGenerator` variable. The class this object is instantiated from is found in the .NET class library and is called `System.Random()`. Line 35 declares the `randomNumberGenerator` to contain an object of type `System.Random`.

```
35:     private System.Random randomNumberGenerator;
```

Because `randomNumberGenerator` is an instance variable of the `Person` class, it is declared `private`. Note that after this declaration, the `randomNumberGenerator` is still empty; no object has yet been assigned to this variable. We must assign an object to the `randomNumberGenerator` before any random numbers can be generated by this variable; lines 37–40 fulfill this important task by containing a constructor for the `Person` class.

The constructor concept has already been discussed and I won’t drill further into this subject now. It is not important to fully understand the constructor mechanism now, as long as you are aware that the essential part is line 39, where a new object of class `System.Random` is assigned to the `randomNumberGenerator` of any newly created object of class `Person`.

```
37:     public Person()
38:     {
39:         randomNumberGenerator = new System.Random();
40:     }
```

The `NewFloorRequest` defined in lines 42–46 will, when called, generate and return a random number that indicates `passenger`’s next floor request. Line 45 finds a random number between 1 and 30 (specified in the parentheses `.Next(1,30)`). The `return` keyword sends this random number back to the caller which, in this case, is the `InitiateNewFloorRequest` method of the `Elevator` object. The details of the `System.Random` class will not be discussed any further here. If you want to investigate this class further, please use the .NET Framework Documentation.

```
42:     public int NewFloorRequest()
43:     {
44:         // Return randomly generated number
45:         return randomNumberGenerator.Next(1,30);
46:     }
```



Note

The programmer who implements the `Elevator` class makes use of the `NewFloorRequest` method of a `Person` object. However, he or she should not need to look at the method definition to use the method. He or she would most likely not be interested in how a `Person` object decides on the values returned and would happily be unaware of random number generators and the like. The method header and a short description of the intent of the method is all that he or she needs to know. Information of how the intent is accomplished has no place here.

The method header forms a contract with the user of the method. It gives the name of the method and how many arguments should be sent along with the method call. The method then promises to return either no value (if stated `void`) or one value of a specific type.

So as long as we don't tinker with the method header (don't change the name of the method, the number and type of its formal parameters, or its return type), we can create all sorts of intricate processes for the `Person` object to decide on the next destination floor.

The `Main()` Method: Conducting the Simulation

In lines 53–63, we find our old friend the unavoidable `Main` method.

```
53:     public static void Main()
54:     {
55:         elevatorA = new Elevator();
56:         elevatorA.LoadPassenger();
57:         elevatorA.InitiateNewFloorRequest();
58:         elevatorA.InitiateNewFloorRequest();
59:         elevatorA.InitiateNewFloorRequest();
60:         elevatorA.InitiateNewFloorRequest();
61:         elevatorA.InitiateNewFloorRequest();
62:         elevatorA.ReportStatistic();
63:     }
```

Even if `Main()` is positioned at the end of the program in this case, it contains, as always, the statements that are to be executed first when the program is started. You can view `Main()` to be a “control tower” of the whole program. It uses the overall functionality provided by the other classes in the program to direct the general flow of execution.

Briefly, the `Main` method initially creates an `Elevator` object and assigns it to `elevatorA` (line 55); instructs it to load in a passenger (line 56); asks `elevatorA` to request a “next floor request” along with its fulfillment five times (lines 57–61); and finally, it asks the `Elevator` object to report the statistic it has collected over those five trips (line 62).

Note how relatively easy it is to conduct this simulation from the `Main` method. All the hard detailed work is performed in the `Elevator` and `Person` classes.

Class Relationships and UML

The three user-defined classes, along with the `System.Random` class of Listing 5.1, collaborate to provide the functionality of our simple elevator simulation program. The `Building` class contains an `Elevator` object and calls its methods, the `Elevator` object employs a `Person` object to direct its movements, and the `Person` object uses a `System.Random` object to decide

the next floor. In well-constructed, object-oriented programs, classes collaborate in similar ways—each contributing with its own unique functionality—for the whole program to work.

If two classes are to collaborate, they must have a relationship (interact) with each other. Several different relationships can exist between two classes, and it is up to the designer of the program to decide which particular relationships should be implemented for the program at hand. This design phase typically takes place when the classes of the program have been identified (stage 2b of the design process described in Chapter 2, “Your First C# Program”). The following section discusses a few commonly found relationships.

`Building-Elevator` and `Elevator-Person` form two kinds of relationships that we will use as examples.

The Building-Elevator Relationship

A typical building is composed of many different parts, such as floors, walls, ceilings, a roof, and sometimes elevators. In our case, we can say that the `Building` we are simulating has an `Elevator` as one of its parts. This is sometimes referred to as a whole/part relationship. We have implemented this relationship in Listing 5.1 by declaring an instance variable of type `Elevator` inside the `Building` class as in line 51:

```
51:    private static Elevator elevatorA;
```

This allows `Building` to hold an `Elevator` object and call its `public` methods. A class can have many different instance variables holding many objects of different classes. For example, we could also have equipped `Building` with instance variables representing a number of `Floor` objects. This concept of constructing a class (`Building`) with other classes (`Elevator` or `Floors` or both and many others) is generally called *aggregation*, and the accompanying relationships *aggregation relationships*.

If the aggregation relationship (as in the case of the `Building-Elevator` relationship), is reflecting a situation where one class is an integral part of the other, we can further call this aggregation a *composition*. (In a moment you will see why the `Elevator-Person` relationship is an aggregation but not a composition). The composition relationship can be illustrated with a Unified Modeling Language (UML) class diagram as shown in Figure 5.3. The two rectangular boxes symbolize classes and the line connecting them with the black diamond (pointing at the whole class) illustrates a composition relationship between the classes. Both classes are marked with the number 1 to indicate that one `Building` has one `Elevator`.



The Unified Modeling Language (UML): The Lingua Franca of OO Modeling

Pseudocode is a useful aid in expressing algorithms that are implemented in single methods because they are read from top to bottom, like the runtime executes the program, and because it abstracts away from the hard-coded rigor of a computer language (semicolons, parentheses, and so on).

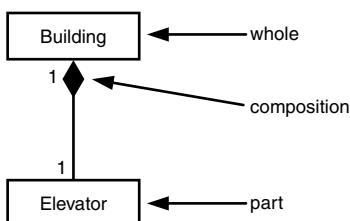
However, classes can consist of many methods, and larger programs consist of many classes.

Pseudocode is not a suitable tool to illustrate models of how the classes of a program relate, because classes break free of the sequential procedural-oriented way of thinking (every class can potentially have a relationship with any other class defined in the program) and because the format of the pseudocode is much too detailed to provide an overview of a large OO program.

To model class relationships and the overall architecture of an OO program effectively, we need a language that allows us to abstract away from the internal details of methods and, instead, provide us with the means to express class relationships and OO concepts on a suitable level of detail. For this purpose, most OO programmers today, irrespective of their programming language, use a graphical diagramming language called the Unified Modeling Language (UML). UML is a feature-rich language and requires a whole book to be amply presented; accordingly, this book only presents a small subset of UML.

You can get detailed information about UML from the non-profit organization Object Management Group (OMG) (www.omg.org) at www.omg.org/uml. Many good UML books have been written including *The Unified Modeling Language User Guide* by the originators of UML, Grady Booch, James Rumbaugh, and Ivar Jacobson.

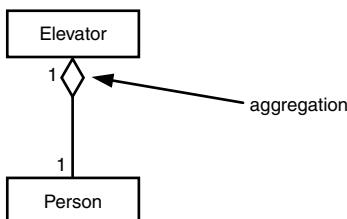
FIGURE 5.3
UML diagram symbolizing composition.



The Elevator-Person Relationship

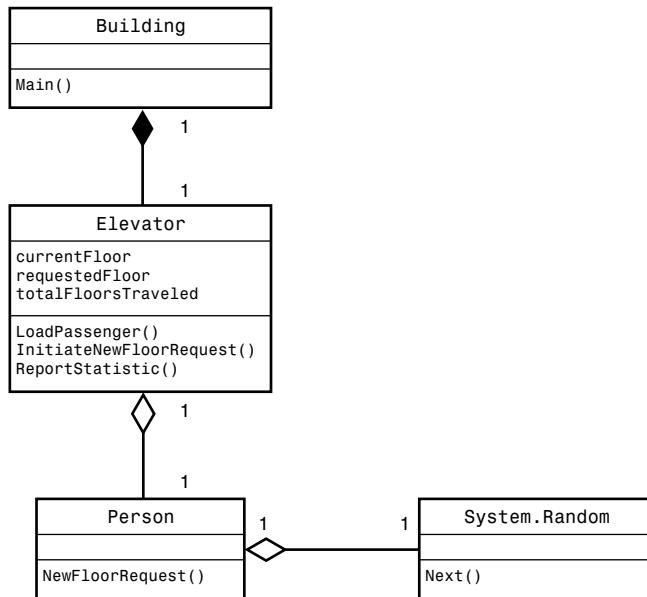
A button is an integral part of an elevator, but a passenger is not. (The elevator is still fully operational without a passenger, but not without its buttons). So even though the passenger in our implementation (for abstraction reasons) has been made a permanent part of the `Elevator` (the same `Person` object stays inside the `Elevator` throughout the simulation), it is not a composition relationship, merely an aggregation. The relationship is illustrated with UML in Figure 5.4. Notice that an open diamond, in contrast to the filled diamond in Figure 5.3, symbolizes aggregation.

FIGURE 5.4
UML diagram symbolizing aggregation.



The whole UML class diagram for the program in Listing 5.1 is shown in Figure 5.5. UML allows us, as shown, to divide the rectangle representing a class into three compartments—the upper compartment contains the name of the class, the middle compartment the instance variables (or attributes), and the lower compartment the methods (behavior) belonging to the class.

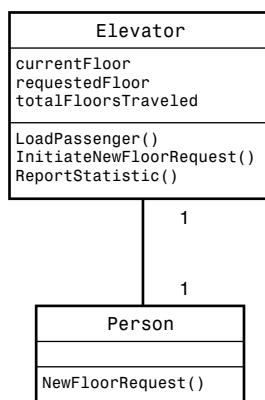
FIGURE 5.5
UML class diagram for Listing 5.1.



Associations

Permanent relationships between classes, such as the aggregation and composition relationships discussed in the previous sections, are generally called *structural relationships* or, more formally, *associations*. However, other types of associations exist that are not aggregations (and therefore not compositions either). To give an example of the latter, consider the following scenario: In an attempt to make our elevator simulation more realistic, we change our original list of abstractions shown earlier by allowing many **Person** objects to enter, travel, and exit the **Elevator** object instead of just the one **Person** object that permanently stays there at the moment. Any one **Person** object could then no longer be said to be a permanent part of the **Elevator** object, and we would just say that the **Elevator** class is associated with the **Person** class. An association is shown with a simple line as in Figure 5.6.

FIGURE 5.6
Elevator/Person class association.



Other examples of associations that are not aggregations are

- Employee works for Company
- BankCustomer interacts with the BankTeller



Note

Associations that precisely connect two classes are called binary associations; they are the most common kind of association.

Summary

This chapter consists of two main parts. The first part is about the lexical structure of a C# source program. The second part provides an example of an object-oriented program, which relates directly to the discussion in Chapter 3 about abstraction and encapsulation.

The following are the most important points covered in this chapter:

A C# source program can be viewed as a collection of identifiers, keywords, whitespace, comments, literals, operators, and separators.

C# is a case-sensitive language. To improve the clarity for other readers of the code, it is important to adhere to a certain capitalization style. Pascal Casing (`ThisIsPascalCasing`) and Camel Casing (`thisIsCamelCasing`) are the preferred styles and are used for different C# constructs.

A literal has the value that is written in the source code (what you see is what you get). Values like 10 and “This is a dog” are examples of literals.

Separators, such as semicolons (;), commas (,), and periods (.), separate different elements in C# from each other.

Operators act on operands. Operands combine with operators to form expressions.

Instance variables must be initialized when an object is created. This is either done automatically by the runtime, by a declaration initialization, or by an instance constructor.

An object can hold a reference to another object in an instance variable. Such a permanent relationship is called an association.

An object is created by using the `new` keyword.

In an object-oriented program, classes collaborate to provide the functionality of the program.

Two classes can collaborate by having a relationship.

Common association relationships are aggregations and compositions.

The Unified Modeling Language (UML) is by far the most popular graphical modeling language used to express and illustrate object-oriented program designs.

Review Questions

1. What is lexical analysis?
2. What are the atomic parts of a C# program?
3. What is Pascal casing and camel casing? For which parts of the C# program should they be used?
4. What is the main difference between variables and literals?
5. What are operators and operands? How do they relate?
6. Is `50` an expression? Is `(50 + x)`? Is `public`?
7. Give examples of typical keywords in C#.
8. Why is pseudocode not well suited for expressing the overall design of an object-oriented program?
9. What kind of relationship does a `BankCustomer` object have with a `BankTeller` object? How is this expressed in UML?
10. What kind of relationship does a `Heart` object have with a `HumanBody` object. How is this expressed in UML?
11. What kind of relationship does a `LightBulb` have with a `Lamp` object. How is this expressed in UML?
12. How is an association relationship implemented in C#?
13. How can instance variables be initialized when an object is created?
14. Describe `passenger` when declared as in the following line:

```
private Person passenger;
```

Programming Exercises

Enable the program in Listing 5.1 to exhibit the following functionality by changing its source code:

1. Print “The simulation has commenced” on the command console right after the program is started.
2. Print “The simulation has ended” as the last thing just before the program is terminated.
3. Instead of merely choosing floors between 1 and 30, the `Person` class chooses floors between 0 and 50.
4. On its first ride, `Elevator` starts at floor number 0 instead of floor number 1.
5. The `Elevator` object does 10 journeys instead of the 5 it is doing now.

6. `Elevator` is currently counting the total floors traveled with the `totalFloorsTraveled` variable. Declare an instance variable inside the `Elevator` class that keeps track of the number of trips this elevator completes. You could call this variable `totalTripsTraveled`. The `Elevator` should update this variable by adding one to `totalTripsTraveled` after every trip. Update the `ReportStatistics` method of the `Elevator` class to print out not only `totalFloorsTraveled` but also `totalTripsTraveled`, accompanied by an explanation for what is being printed.
7. Add an instance variable to the `Elevator` class that can hold the name of the elevator. You can call it `myName`. Which type should you use for this instance variable? Should you use `private` or `public` when you declare it? Write a constructor by which you can set the value of this variable as you create the `Elevator` object with `new` and assign it to a variable. (Hint: The constructor is a method and must have the same name as its class. This constructor must have a formal parameter of type `string` in its header.) Adjust the call to the constructor when using the keyword `new` by inserting the name of the `Elevator` as an argument (between the parenthesis—instead of `new Elevator()`, write `new Elevator("ElevatorA")`).

Every time the `Elevator` completes a trip, it should print its name along with its departing and arrival floor. In other words, instead of printing

`Departing floor: 2 Traveling to floor: 24`

it should print

`ElevatorA: Departing floor: 2 Traveling to floor: 24`

where `ElevatorA` is the name of the `Elevator` residing inside `myName`.

Introduction to Windows Forms

CHAPTER

3.1

IN THIS CHAPTER

- The Hello Windows Forms Application 189
- Creating and Using an Event Handler 192
- Defining the Border Style of the Form 195
- Adding a Menu 196
- Adding a Menu Shortcut 198
- Handling Events from Menus 199

Windows Forms is the .NET replacement for MFC. Unlike the MFC library, which was a thin(ish) wrapper on the Win32 API, Windows Forms is a totally object-oriented, hierarchical answer to Windows development under .NET.

Despite its “Forms” epithet, the layout of components is not done with a resource file as is the case in MFC dialogs and form windows. Every component is a concrete instance of a class.

Placement of the components and control of their properties are accomplished by programming them via their methods and accessors. The drag-and-drop tools used to define a Windows Form are actually maintaining the source code that initializes, places, and allows interaction with a target application.

Resource files are used by Windows Forms for tasks such as placing images on the form or storing localized text, but not in the familiar format that has been used by Windows since the 1980s. As you might expect, the new resource file format is an XML file. We’ll examine resources in more detail in Chapter 3.4, “Windows Forms Example Application (Scribble .NET).”

Windows Forms layout can be done by dragging components from a tool palette onto a form. You can use VS.NET for this or alternatively, if you really want the whole hands-on experience, you can lay out your forms by coding the objects directly in C#, VB, managed C++, or any of the .NET languages. To give you the benefit of understanding what really goes on in a Windows Form, we won’t be dealing with any of the design tools early in this section. Rather, we will do as much as possible by hand and then, when you understand the basics, move on to using the drag-and-drop tools.

User interaction in Windows Forms is accomplished through events. The components provide event sources, such as mouse movement, position, and button clicks, and then you wire the events to handlers. These are functions called by a standard form of delegate, which means there are no message maps of which to keep track.

The Hello Windows Forms Application

Listing 3.1.1 is a simple Windows Forms application that displays the text “Hello Windows Forms!” in a label on the main form.

LISTING 3.1.1 HWF.cs: The Hello Windows Forms Application

```
// HWF.cs
namespace HelloWindowsFormsNamespace {

    using System;
    using System.Drawing;
    using System.ComponentModel;
    using System.WinForms;
```

LISTING 3.1.1 Continued

```
public class HelloWindowsForms : System.WinForms.Form
{
    //Label 1 displays the text message "Hello Windows Forms!"
    private System.WinForms.Label label1;

    //The constructor is where all initialization happens.
    //Forms created with the designer have an InitializeComponent() method.
    public HelloWindowsForms()
    {

        this.label1 = new System.WinForms.Label();

        label1.Location = new System.Drawing.Point(8, 8);
        label1.Text = "Hello Windows Forms!";
        label1.Size = new System.Drawing.Size(408, 48);
        label1.Font = new System.Drawing.Font("Microsoft Sans Serif", 24f);
        label1.TabIndex = 0;
        label1.TextAlign = System.WinForms.HorizontalAlignment.Center;

        this.Text = "Hello World";
        this.MaximizeBox = false;
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.FormBorderStyle = System.WinForms.FormBorderStyle.FixedDialog;
        this.MinimizeBox = false;
        this.ClientSize = new System.Drawing.Size(426, 55);
        this.Controls.Add(label1);
    }

    // This main function instantiates a new form and runs it.
    public static void Main(string[] args)
    {
        Application.Run(new HelloWindowsForms());
    }
}

} // end of namespace
```

3.1

You can see that the class `HelloWindowsForms` contains a simple label component, `label1`. It has only two methods: a static `Main` method that creates and runs an instance of the `HelloWindowsForms` class and a constructor in which all of the component initialization occurs.

This shows the minimum functionality and minimum complication of the Windows Forms system. Programs designed and maintained by VS.NET or WinDes.exe have some added functions and data members that detract from the simplicity of this example, but are necessary for the RAD environments to keep track of the form design.

To build the executable, we have prepared a small batch file that creates a Windows executable and references all the correct library DLLs. Using this batch file will allow you to stay away from the Visual Studio build environment for a while—because it hides too much of the important information—and concentrate on the actual functionality. Listing 3.1.2 shows build.bat.

LISTING 3.1.2 build.bat

```
csc /t:winexe /r:Microsoft.win32.Interop.dll, system.dll,  
➥system.configuration.dll, system.data.dll,  
➥system.diagnostics.dll, system.drawing.dll,  
➥ system.winforms.dll /out:%1.exe %1.cs  
➥ %2 %3 %4 %5 %6 %7 %8 %9
```

build.bat is available with the software that accompanies this book, but if you want to type it in yourself, make sure that it's all on the same line and there are no spaces between the libraries named in the /r: reference directive. To use build.bat, simply type **build** followed by the name of the .cs file to compile. For example

build HWF

will compile the **HWF.cs** file and create **HWF.exe**.

Running the **h wf.exe** program will produce a form shown in Figure 3.1.1.



FIGURE 3.1.1

The Hello Windows Forms Application.

Creating and Using an Event Handler

If you're familiar with Windows C development, you will understand the idea of handling messages as they arrive by the action of the `WndProc` switch statement. If you have been working with MFC, you will know that messages are routed through the message map of your application. Under .NET the message routing system is provided by delegates. Your application will define event handlers that are called through a delegate whenever a message arrives. Button clicks for example, can be handled by using a generic `EventHandler` delegate. Other types of events, such as mouse events or timer events, are serviced through specific handlers that have their own delegate signatures.

Adding a button to the Windows Forms application is simple. You can add a button member to the class, initialize it in the constructor, and place it in the controls collection.

When this is done, you can wire up an event that is trapped whenever the button is clicked. Events sent through the system are hooked to event handlers of a specific signature by a multi-cast delegate that is declared especially for this purpose. Your event handlers must all have the following method signature:

```
void EventMethodName(Object sender, EventArgs e);
```

Event handlers are added to the control's event sources with the `+=` operator and removed with the `-=` operator, so events can be modified dynamically at runtime. Because events are multi-cast, you can add more than one handler at a time to the same event.

Listing 3.1.3 shows the modified file `HWFButton.cs`.

LISTING 3.1.3 HWFButton.cs: Hello Windows Forms with a Simple Button Handler

```
// HWFButton.cs
namespace HelloWindowsFormsNamespace {

    using System;
    using System.Drawing;
    using System.ComponentModel;
    using System.WinForms;

    public class HelloWindowsForms : System.WinForms.Form
    {

        //Label 1 displays the text message "Hello Windows Forms!"
        private System.WinForms.Label label1;

        //Adding a button member allows us to place a button on the panel.
        private System.WinForms.Button button1;

        //The constructor is where all initialization happens.
        //Forms created with the designer have an InitializeComponent() method.
        public HelloWindowsForms()
        {

            this.label1 = new System.WinForms.Label();

            label1.Location = new System.Drawing.Point(8, 8);
            label1.Text = "Hello Windows Forms!";
            label1.Size = new System.Drawing.Size(408, 48);


```

LISTING 3.1.3 Continued

```
label1.Font = new System.Drawing.Font("Microsoft Sans Serif", 24f);
label1.TabIndex = 0;
label1.TextAlign = System.WinForms.HorizontalAlignment.Center;

this.button1=new System.WinForms.Button();

button1.Location=new System.Drawing.Point(8,58);
button1.Size = new System.Drawing.Size(408,25);
button1.Text = "Click Me!";
button1.TabIndex = 1;
button1.Click += new EventHandler(OnButton1Clicked);

    this.Text = "Hello World";
    this.MaximizeBox = false;
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.FormBorderStyle = System.WinForms.FormBorderStyle.FixedDialog;
    this.MinimizeBox = false;
    this.ClientSize = new System.Drawing.Size(426, 85);

    this.Controls.Add(label1);
this.Controls.Add(button1);
}

void OnButton1Clicked(Object sender,EventArgs e)
{
if(this.label1.Text == "Hello Windows Forms!")
    this.label1.Text = "You Clicked?";
else
    this.label1.Text = "Hello Windows Forms!";
}

// This main function instantiates a new form and runs it.
public static void Main(string[] args)
{
    Application.Run(new HelloWindowsForms());
}
}

} // end of namespace
```

Type in Listing 3.1.3 and build it with the build batch file. Running the sample will show a window that has the same message and a clickable button. The button handler simply swaps the text in the label control to verify that the handler worked.

NOTE

Note that the code in Listing 3.1.3 uses the fully qualified name for the components and methods:

```
button1.Size = new System.Drawing.Size(408,25);
```

This longhand form is not always necessary. You could abbreviate the line above to

```
button1.Size = new Size(408,25);
```

Layout tools will always give you the longhand version because they never get tired of typing.

Generally, if the object you're creating is in a namespace you declared you were using, you can use the shorthand.

When you run the file, you will see the application shown in Figure 3.1.2.

**3.1**

INTRODUCTION TO
WINDOWS FORMS

FIGURE 3.1.2

Using a simple Click Handler.

Defining the Border Style of the Form

The previous examples are both simple, fixed-size forms that have no minimize or restore button. The border style of the form object controls how a form is shown and if it can be resized.

Listing 3.1.4 shows a very simple Windows Forms application with a sizable client area.

LISTING 3.1.4 `resize.cs`: A Simple Resizable Windows Forms Application

```
using System;
using System.Drawing;
using System.ComponentModel;
using System.WinForms;
```

LISTING 3.1.4 Continued

```
public class SizeApp : System.WinForms.Form
{
    public     SizeApp()
    {
        this.Text = "SizeApp";
        this.MaximizeBox = true;
        this.FormBorderStyle = FormBorderStyle.Sizable;
    }

    static void Main()
    {
        Application.Run(new SizeApp());
    }
}
```

Building and running this application will result in a resizable application that can also be minimized to the taskbar and restored in the normal way.

Adding a Menu

A Windows application without a menu is a rare thing. A Windows Forms application is no exception. Like the button and label you saw earlier, the menu component can be added to the `Menu` member of the main application, and events from the menu items can be hooked to handlers.

Menus under .NET come in two forms. `MainMenu` is applied to a form to provide the main user interface menu, and `ContextMenu` is used to respond to right mouse clicks. In both cases, the individual items within the menus are objects of type `MenuItem`. A menu is constructed as a hierarchy of parent and child objects. The main menu owns the individual drop-downs, which in turn own their menu items.

A typical menu creation sequence is seen in Listing 3.1.5.

LISTING 3.1.5 Constructing a Menu

```
MainMenu menu = new MainMenu();

MenuItem filemenu = new MenuItem();
filemenu.Text = "File";
menu.MenuItems.Add(filemenu);
MenuItem open = new MenuItem();
open.Text = "Open";
filemenu.MenuItems.Add(open);
```

LISTING 3.1.5 Continued

```
MenuItem save= new MenuItem();
save.Text = "Save";
filemenu.MenuItems.Add(save);

MenuItem exit= new MenuItem();
exit.Text = "Exit";
filemenu.MenuItems.Add(exit);

MenuItem editmenu = new MenuItem();
editmenu.Text = "Edit";
menu.MenuItems.Add(editmenu);

MenuItem cut= new MenuItem();
cut.Text = "Cut";
editmenu.MenuItems.Add(cut);

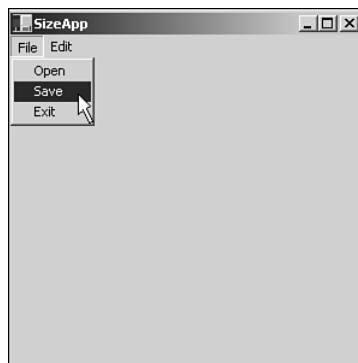
MenuItem copy = new MenuItem();
copy.Text = "Copy";
editmenu.MenuItems.Add(copy);

MenuItem paste = new MenuItem();
paste.Text = "Paste";
editmenu.MenuItems.Add(paste);

this.Menu = menu;
```

The indentation in Listing 3.1.5 illustrates the hierarchy of the menus.

Figure 3.1.3 shows a simple resizable application with a menu added.

**3.1****FIGURE 3.1.3**

A simple resizable application with a menu.

Adding a Menu Shortcut

Placing an ampersand before a character in the menu text will automatically give the menu item an underscore when the Alt key is pressed. The key combination of Alt+F followed by O can be used to invoke the menu handler as if the menu were selected with the mouse.

A direct key combination might also be added to the menu item by using one of the predefined Shortcut enumerations. The File, Open menu item handler can be made to fire in response to a Ctrl+O keypress by adding the shortcut, as shown in Listing 3.1.6.

LISTING 3.1.6 Adding a Shortcut to the File, Open MenuItem

```
MenuItem filemenu = new MenuItem();
filemenu.Text = "&File";
menu.MenuItems.Add(filemenu);
MenuItem open = new MenuItem();
open.Text = "&Open";
filemenu.MenuItems.Add(open);
open.Shortcut = Shortcut.CtrlO;
open.ShowShortcut = true;
```

When you press the Alt key, the F in the File menu is underlined. You can press **F** to pop up the menu and press **o** to invoke the menu's function, as shown in Figure 3.1.4.



FIGURE 3.1.4

Menu shortcuts in action.

Note how the Open menu is appended with the shortcut key press combination Ctrl+O by the MenuItem.ShowShortcut property setting.

Handling Events from Menus

There are several events that you can handle from a `MenuItem`. The most important is the one that you defined the item for in the first place. Remember from our button click example earlier that the events are hooked to the `Click` event source by a delegate defined by the system. In Listing 3.1.7, a handler that pops up a message box is defined and added to the methods of the `SizeApp` class.

LISTING 3.1.7 A Simple Event Handler

```
public class SizeApp : System.WinForms.Form
{
    public void OnFileOpen(Object sender, EventArgs e)
    {
        MessageBox.Show("You selected File-Open!");
    }
}
```

The event handler has the standard method signature for events `void_function(Object, EventArgs)` and can be added to the File, Open `MenuItem`'s `Click` event source like this:

```
open.Click += new EventHandler(OnFileOpen);
```

This line is added to the menu setup code after the shortcut initialization.

Whenever the menu item is selected with the mouse, the Alt+F+O key sequence, or the Ctrl+O shortcut, the menu handler will fire and the message box will pop up.

The complete C# file for the modified `resize.cs` program is available as `resize2.cs` on this book's Web site. You can find this at <http://www.samspublishing.com/>. Type in the ISBN **067232153X** for this book.

3.1

INTRODUCTION TO
WINDOWS FORMS

User Interface Control Events for `MenuItem`s

Other events are fired by `MenuItem`s to enable you to give better feedback to the user or to customize the user experience. MFC had the `CCmdUI` class for this purpose. Windows Forms provides the `Popup` event source.

Just before a `MenuItem` is shown, the `Popup` event is fired to give you time to decide whether to show, check, or change the appearance of a menu item. You can trap this event by adding an event handler to the `Popup` event source:

```
filemenu.Popup += new EventHandler(OnPopupMenu);
```

The handler for this event is defined in Listing 3.1.8. It shows some of the standard things you can do, checking, enabling, hiding, and so on, with `MenuItem`s. The class has a Boolean variable called `m_bPopupChecked`. Every time the File menu is expanded, the program toggles this variable to true or false depending on its previous state. The `Sender` object is known to be a `MenuItem`, so it's possible to cast to that type safely. The three menu entries in the File menu are then checked, disabled, or hidden entirely, depending on the state of the variable. The image (seen in Figure 3.1.5) shows the menus in their two states.

LISTING 3.1.8 The Popup Event Handler

```
bool m_bPopupChecked;

public void OnPopupFilemenu(Object Sender, EventArgs e)
{
    // this handler illustrates the Popup event and the MenuItem UI properties.
    m_bPopupChecked = !m_bPopupChecked;
    MenuItem item = (MenuItem)Sender;
    item.MenuItems[0].Checked = m_bPopupChecked;
    item.MenuItems[1].Enabled = m_bPopupChecked;
    item.MenuItems[2].Visible = m_bPopupChecked;
}
```



FIGURE 3.1.5

The menu after the Popup event.

Defining a `MenuItem` as a Separator

Very often a group of menu entries will be strongly associated with one another, or one menu item will be separated from another by strategic placement of a menu separator. Under Windows Forms the menu separator is a menu item that does nothing but draw a line across the menu. This is very simple; just set the text of a menu item to a single dash:

```
MenuItem dummysmenu = new MenuItem();
dummysmenu.Text = "Separator";
menu.MenuItems.Add(dummysmenu);
    dummysmenu.MenuItems.Add(new MenuItem("Above"));
    dummysmenu.MenuItems.Add(new MenuItem("-"));
    dummysmenu.MenuItems.Add(new MenuItem("Below"));
```

Handling the Select Event

Once a menu item has been popped up, all of its visible members can be selected by positioning the mouse over them or using the arrows keys. When this selection takes place, an event is fired. The event source for this is called `Select`, and it is handled in much the same way as the `Popup` event.

The `Select` event is used primarily to update a status bar or other control with a help string that explains an otherwise cryptic menu entry. It could also be used for other user-interface customization.

The demonstration in Listing 3.1.9 uses the `Select` event to display a string in a label control on the client area.

3.1

INTRODUCTION TO
WINDOWS FORMS

LISTING 3.1.9 menus.cs: Handling the Select Event

```
using System;
using System.Drawing;
using System.ComponentModel;
using System.WinForms;

public class menuapp : System.WinForms.Form
{
    Label label;

    void ShowInfo(Object Sender,EventArgs e)
    {
        MenuItem item=(MenuItem)Sender;
        switch(item.Text)
        {
            case "&Open":
                label.Text = "Open a file from disk";
                break;
            case "&Save":
                label.Text = "Save a file onto disk";
                break;
        }
    }
}
```

LISTING 3.1.9 Continued

```
        case "E&xit":
            label.Text = "Exit MenuApp";
            break;
    }
}

public menuapp()
{
    this.Text = "MenuApp";
    this.MaximizeBox = true;
    this.BorderStyle = FormBorderStyle.Sizable;

    this.label = new Label();
    label.Location = new Point(8,100);
    label.Size = new Size(200,25);

    this.Controls.Add(label);

    MainMenu menu = new MainMenu();

    MenuItem filemenu = new MenuItem();
    filemenu.Text = "&File";
    menu.MenuItems.Add(filemenu);

    MenuItem open = new MenuItem();
    open.Text = "&Open";
    open.Select += new EventHandler>ShowInfo());
    filemenu.MenuItems.Add(open);

    MenuItem save= new MenuItem();
    save.Text = "&Save";
    save.Select += new EventHandler>ShowInfo());
    filemenu.MenuItems.Add(save);

    MenuItem exit= new MenuItem();
    exit.Text = "E&xit";
    exit.Select += new EventHandler>ShowInfo());
    filemenu.MenuItems.Add(exit);

    this.Menu = menu;
}

static void Main()
```

LISTING 3.1.9 Continued

```
{  
    Application.Run(new menuapp());  
}  
}
```

Figure 3.1.6 shows the `menus.cs` program in action.

**3.1**

INTRODUCTION TO
WINDOWS FORMS

FIGURE 3.1.6

The Select event handler in action.

Menu Layout

Menus are built up from `MenuItem` components. These can be arranged across the screen on the menu bar, and are most often arranged vertically in drop-down menus. You can change the default layout of `MenuItem`s to give a different UI style.

The `Break` and `BarBreak` methods are used to create menus that are arranged horizontally rather than vertically. Setting the `BarBreak` property in a `MenuItem` causes the item to be drawn in a new column. `BarBreak` adds a vertical separator bar to the menu between the columns. `Break` makes a new column but doesn't add the vertical bar. The modifications to the `menus.cs` code on lines 14 and 20 in the following result in the change seen in Figure 3.1.7.

```
1:     MenuItem filemenu = new MenuItem();  
2:     filemenu.Text = "&File";  
3:     menu.MenuItems.Add(filemenu);  
4:  
5:     MenuItem open = new MenuItem();  
6:     open.Text = "&Open";  
7:     open.Select += new EventHandler>ShowInfo);  
8:     filemenu.MenuItems.Add(open);
```

```
9:  
10:    MenuItem save= new MenuItem();  
11:    save.Text = "&Save";  
12:    save.Select += new EventHandler(ShowInfo);  
13:    filmenu.MenuItems.Add(save);  
14:    save.BarBreak=true;  
15:  
16:    MenuItem exit= new MenuItem();  
17:    exit.Text = "E&xit";  
18:    exit.Select += new EventHandler(ShowInfo);  
19:    filmenu.MenuItems.Add(exit);  
20:    exit.Break=true;  
21:
```

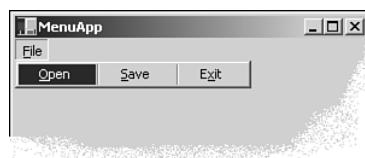


FIGURE 3.1.7

The BarBreak property.

Similarly, the code changes to lines 14 and 20 in the following result in the menu style shown in Figure 3.1.8.

```
1:    MenuItem filmenu = new MenuItem();  
2:    filmenu.Text = "&File";  
3:    menu.MenuItems.Add(filmenu);  
4:  
5:    MenuItem open = new MenuItem();  
6:    open.Text = "&Open";  
7:    open.Select += new EventHandler(ShowInfo);  
8:    filmenu.MenuItems.Add(open);  
9:  
10:   MenuItem save= new MenuItem();  
11:   save.Text = "&Save";  
12:   save.Select += new EventHandler(ShowInfo);  
13:   filmenu.MenuItems.Add(save);  
14:   //save.BarBreak=true;  
15:  
16:   MenuItem exit= new MenuItem();  
17:   exit.Text = "E&xit";  
18:   exit.Select += new EventHandler(ShowInfo);  
19:   filmenu.MenuItems.Add(exit);  
20:   exit.Break=true;  
21:
```

**FIGURE 3.1.8**

The Break Property in use.

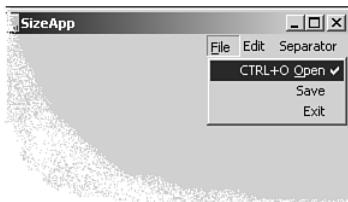
Each time you set the `Break` property, the `MenuItem` is placed in a new column.

Right-to-Left Menus

To cater to cultures that read right-to-left or to add an unconventional style to your menus, you can modify the menu's `RightToLeft` property.

```
1: MainMenu menu = new MainMenu();
2: menu.RightToLeft=RightToLeft.Yes;
3: MenuItem filemenu = new MenuItem();
4: filemenu.Text = "&File";
```

Adding line 2 to `resize.cs` results in the effect seen in Figure 3.1.9.

**3.1**

INTRODUCTION TO
WINDOWS FORMS

FIGURE 3.1.9

Right-to-left reading menus.

Creating and Using a Context Menu

A context menu is a floating menu that can pop up wherever it's needed for selections in a particular editing or user interface context. The convention is for the context menu to appear in response to a right-mouse button click.

Windows Forms follows this convention. The application has a `ContextMenu` property that, like its `MainMenu` counterpart, is available from the top-level application form.

Adding, showing, or modifying the appearance of a context menu and specifying its handler are largely the same as the main menu. The following adds a context menu with three items to the main window of the `resize.cs` form:

```
ContextMenu cmenu = new ContextMenu();
cmenu.MenuItems.Add(new MenuItem("&First"));
cmenu.MenuItems.Add(new MenuItem("&Second"));
cmenu.MenuItems.Add(new MenuItem("-"));
cmenu.MenuItems.Add(new MenuItem("&Third"));
this.ContextMenu=cmenu;
```

A simple right mouse click will bring up the context menu.

Replacing, Cloning, and Merging Menus

A common practice in Windows applications is to change menus according to the data displayed in the client window. To save you from having to create and destroy menus on-the-fly, .NET allows you to swap out menus and create new ones by merging one or more predefined menu hierarchies.

Listing 3.1.10 shows a simple application that constructs several menus and then combines them in different ways according to selections made by the user.

LISTING 3.1.10 menuswop.cs: Manipulating, Cloning, and Merging Menus

```
1: using System;
2: using System.Drawing;
3: using System.ComponentModel;
4: using System.WinForms;
5:
6: namespace Sams {
7:
8: class menuswop : System.WinForms.Form
9: {
10:    MainMenu m_menu;
11:    MenuItem m_editmenu,m_menumenu,m_switchitem,m_showsecond,m_merge;
12:    MenuItem m_playmenu;
13:
14:    bool m_bswop;
15:    bool m_bshowsecond;
16:    bool m_bmerge;
17:
18:    // private helper function for the BuildMenu function.
19:    void addmenuItem(MenuItem menu, string s)
20:    {
21:        MenuItem temp=new MenuItem(s);
22:        temp.Enabled= false;
23:        menu.MenuItems.Add(temp);
24:    }
25:
```

LISTING 3.1.10 Continued

```
26: // This builds a menu structure from copies
27: // of the originals using CloneMenu.
28: void BuildMenu()
29: {
30:     m_menu=new MainMenu();
31:     m_menu.MenuItems.Add(m_menumenu.CloneMenu());
32:
33:     if(m_bmerge) // when we merge...
34:     {
35:         MenuItem temp=new MenuItem();
36:
37:         if(!m_bswop)
38:         {
39:             addMenuItem(temp,"Edit");
40:             temp.MergeMenu(m_editmenu.CloneMenu());
41:         }
42:         else
43:         {
44:             addMenuItem(temp,"Play");
45:             temp.MergeMenu(m_playmenu.CloneMenu());
46:         }
47:
48:         temp.MenuItems.Add(new MenuItem("-"));
49:
50:         if(m_bshowsecond)
51:         {
52:             if(!m_bswop)
53:             {
54:                 addMenuItem(temp,"Play");
55:                 temp.MergeMenu(m_playmenu.CloneMenu());
56:             }
57:             else
58:             {
59:                 addMenuItem(temp,"Edit");
60:                 temp.MergeMenu(m_editmenu.CloneMenu());
61:             }
62:         }
63:
64:         temp.Text = "&Merged";
65:         m_menu.MenuItems.Add(temp);
66:
67:     }
68:     else // when we dont merge...
69:     {
```

LISTING 3.1.10 Continued

```
70:         if(!m_bswop)
71:         {
72:             if(m_bshowsecond)
73:             {
74:                 m_menu.MenuItems.Add(m_editmenu.CloneMenu());
75:             }
76:             m_menu.MenuItems.Add(m_playmenu.CloneMenu());
77:         }
78:         else
79:         {
80:             if(m_bshowsecond)
81:             {
82:                 m_menu.MenuItems.Add(m_playmenu.CloneMenu());
83:             }
84:             m_menu.MenuItems.Add(m_editmenu.CloneMenu());
85:         }
86:     }
87:
88:     this.Menu = m_menu;
89: }
90:
91: //This method sets or resets the checks on menu items
92: //note how the MenuItem collection is accessible as an array.
93: void PopupMenuMenu(Object sender, EventArgs e)
94: {
95:     m_menu.MenuItems[0].MenuItems[0].Checked = m_bswop;
96:     m_menu.MenuItems[0].MenuItems[1].Checked = m_bshowsecond;
97:     m_menu.MenuItems[0].MenuItems[2].Checked = m_bmerge;
98: }
99:
100: // The event handler for the switch menu entry
101: void OnSwitchMenu(Object sender, EventArgs e)
102: {
103:     m_bswop = !m_bswop;
104:     BuildMenu();
105: }
106:
107: //The event handler for the show menu entry
108: void Onshowsecond(Object sender, EventArgs e)
109: {
110:     m_bshowsecond = !m_bshowsecond;
111:     BuildMenu();
112: }
113:
```

LISTING 3.1.10 Continued

```
114: //The event handler for the merge menu entry
115: void OnMerge(Object sender, EventArgs e)
116: {
117:     m_bmerge = !m_bmerge;
118:     BuildMenu();
119: }
120:
121: public menuswop()
122: {
123:     // setup a main menu
124:     m_menumenu = new MenuItem("&Menu");
125:     m_menumenu.Popup += new EventHandler(PopupMenuMenu);
126:
127:     //Create the switch item.
128:     m_switchitem=new MenuItem("&Switch");
129:     m_switchitem.Click+=new EventHandler(OnSwitchMenu);
130:     m_menumenu.MenuItems.Add(m_switchitem);
131:
132:     m_showsecond = new MenuItem("&Show");
133:     m_showsecond.Click+= new EventHandler(Onshowsecond);
134:     m_menumenu.MenuItems.Add(m_showsecond);
135:
136:     m_merge = new MenuItem("&Merge");
137:     m_merge.Click += new EventHandler(OnMerge);
138:     m_menumenu.MenuItems.Add(m_merge);
139:
140:     // create a second menu
141:     m_editmenu=new MenuItem("&Edit");
142:     m_editmenu.MenuItems.Add(new MenuItem("Cut"));
143:     m_editmenu.MenuItems.Add(new MenuItem("Copy"));
144:     m_editmenu.MenuItems.Add(new MenuItem("Paste"));
145:
146:     // and an alternative.
147:     m_playmenu=new MenuItem("&Play");
148:     m_playmenu.MenuItems.Add(new MenuItem("Normal"));
149:     m_playmenu.MenuItems.Add(new MenuItem("Fast Forward"));
150:     m_playmenu.MenuItems.Add(new MenuItem("Reverse"));
151:
152:     m_bshowsecond=true;
153:
154:     //Now build the menu from its parts.
155:     BuildMenu();
156:
157: }
```

LISTING 3.1.10 Continued

```
158:  
159:     public static void Main()  
160:     {  
161:         Application.Run(new menuswop());  
162:     }  
163: }  
164:  
165: }// end of Sams namespace
```

In this listing you can see that initially the menu creation process is normal, but that no menus are actually added to the `MenuItem` lists of a parent. The Edit and Play menus are kept separate.

The `BuildMenu` function on line 28 creates a new main menu and assigns it to the application's main menu. This will cause the GC to delete all of the submenus and entries from the main menu. `BuildMenu` then goes on to create the menu order or merge the two submenus into one, depending on the settings of the class member variables.

This technique allows you to create an initial set of menu items with all their handlers and settings in place, and then use them in many combinations without having to keep track of them or reset them to default values.

In Listing 3.1.10 the merge is very simple. There are however, more complex ways that `MenuItem`s can be combined. The `MenuItem` has a merge order property so that when they are merged into another menu, they will sort themselves into a specific sequence. For example, you might want to have a File menu on which functionality is grouped according to the file type or system resources. With the menu merge order, you can ensure that common functions are first in the menu and less common ones are last. To illustrate the merge order functionality, Listing 3.1.11 shows a program in which the menu sorts itself according to the popularity of the selections.

LISTING 3.1.11 menuorder.cs: On-the-Fly Menu Reordering and Merging

```
1: using System;  
2: using System.Drawing;  
3: using System.ComponentModel;  
4: using System.WinForms;  
5:  
6: namespace Sams {  
7: class menuorder : System.WinForms.Form  
8: {  
9:     MainMenu m_menu;  
10:    MenuItem m_workingmenu;  
11:    MenuItem m_menuentry;
```

LISTING 3.1.11 Continued

```
12:
13:    // this event handler is called whenever an item is used
14:    void OnUsed(Object sender, EventArgs e)
15:    {
16:        for(int i=0;i<m_menuentry.MenuItems.Count;i++)
17:            m_menuentry.MenuItems[i].MergeOrder++;
18:
19:        MenuItem m=(MenuItem)sender;
20:
21:        m.MergeOrder--;
22:
23:        if(m.MergeOrder < 0)
24:            m.MergeOrder = 0;
25:
26:    }
27:
28:    // this event handler is also invoked. You could have
29:    // many event handlers attached to the Click sources
30:    void OnClicked(Object sender, EventArgs e)
31:    {
32:        MenuItem m=(MenuItem)sender;
33:        MessageBox.Show("You clicked "+m.Text);
34:    }
35:
36:    //As a menu is popped up it is constructed on the fly.
37:    void OnPopup(Object sender, EventArgs e)
38:    {
39:        m_menu.MenuItems.Clear();
40:        m_workingmenu.MenuItems.Clear();
41:        m_workingmenu.MergeMenu(m_menuentry);
42:        m_menu.MenuItems.Add(m_workingmenu);
43:    }
44:
45:    // Sets up the initial menu text and orders.
46:    public menuorder()
47:    {
48:
49:        string[] s=new string[8]{"Cats","Dogs","Elephants","Geese",
50:        "Moses","Rats","Guinea-pigs","Horses"};
51:
52:        m_menu = new MainMenu();
53:        m_menuentry = new MenuItem("&Menu");
54:        m_menuentry.Popup += new EventHandler(OnPopup);
55:        m_workingmenu = new MenuItem();
```

LISTING 3.1.11 Continued

```
56:
57:         for(int i=0;i<8;i++)
58:         {
59:             MenuItem temp = new MenuItem(s[i]);
60:             temp.MergeOrder=i;
61:             temp.Click+=new EventHandler(OnUsed);
62:             temp.Click+=new EventHandler(OnClicked);
63:             m_menuentry.MenuItems.Add(temp);
64:         }
65:
66:         m_workingmenu.MergeMenu(m_menuentry);
67:         m_menu.MenuItems.Add(m_workingmenu);
68:         this.Menu = m_menu;
69:     }
70:
71: // instantiates and runs the application.
72: public static void Main()
73: {
74:     Application.Run(new menuorder());
75: }
76: }
77:
78: }
```

This listing also illustrates how you can add more than one event handler to a Windows Forms Click. The same is true for the Select, Popup, and all other events.

Adding Sub Menus

MenuItem objects in Windows Forms have their own MenuItem collections. This allows you to create a hierarchical structure of menus that have their own cascading child menus within them. The following listing illustrates this process.

```
MenuItem filemenu = new MenuItem();
filemenu.Text = "&File";
menu.MenuItems.Add(filemenu);

MenuItem open = new MenuItem();
open.Text = "&Open";
filemenu.MenuItems.Add(open);

MenuItem print= new MenuItem();
print.Text = "Print...";
filemenu.MenuItems.Add(print);
```

```
MenuItem temp= new MenuItem();
temp.Text = "Pre&view";
print.MenuItems.Add(temp);

temp= new MenuItem();
temp.Text = "To &File";
print.MenuItems.Add(temp);

MenuItem exit= new MenuItem();
exit.Text = "E&xit";
filemenu.MenuItems.Add(exit);
```

As before, the indentation shows the menu level. Figure 3.1.10 shows the menu in action.

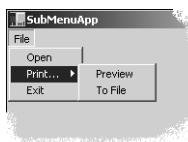


FIGURE 3.1.10

Submenus in action.

3.1

INTRODUCTION TO
WINDOWS FORMS

Summary

In this chapter you have progressed from the simplest Windows Forms application to relatively complex menu and event handler operations. You've seen how the Delegate system replaces the message map or the WndProc for message routing in your application, and you've seen a little of the basic .NET framework for GUI components. Coming up in Chapter 3.2 “User Interface Components,” we'll deal with a selection of the more commonly used Windows Forms components and show you how to lay these items out for use in your forms and dialogs.

User Interface Components

CHAPTER

3.2

IN THIS CHAPTER

- Dialogs 216
- Creating Dialogs 234
- Using Controls 242

In this chapter, we will be showing you a good selection of the standard controls available, how to place and use them, how to add handlers for some of the more useful events, and how to create an application with some nice user interface features.

Windows Forms has a lot of standard user interface components that can be used in your applications. As you might expect, many of them are far more advanced through evolution than their Win32 counterparts. There are dialogs, list boxes, tree controls, panels, labels, toolbars, and many more. There is a large selection of common dialog controls including file selection, color picker, font style, and print dialogs. After a menu format, the dialog box is probably the most widely employed user interface item. Let's examine them more closely now.

Dialogs

Dialogs come in two styles. The *modal* style requires that the user complete the actions on the dialog before returning to the normal flow of the application. *Modeless*, which is a simple fixed size window that can be used to perform some action, but leaves the user free to work on the main application too.

NOTE

Dialogs are best when they are simple. The design of a dialog is crucial to the usability of the application. Too many small dialogs can be annoying, especially if there is no choice of seeing them or not. Large dialogs, especially modal ones, are annoying because they require a lot of interaction that can break the flow of the application.

Dialogs under MFC were the usual odd mixture of Win32 API and object-oriented wrapper code, particularly in the Dialog Data eXchange or DDX system. This represented data in the dialog as data members of the dialog class, but used these members as a staging area for the real information that is edited or selected by the Windows controls on the dialog surface. This meant that there was no instant update mechanism for dialogs because data needed to be transferred to and from the dialog's child windows by the use of Windows messages. Dialogs under Windows Forms are different. They are, like the applications that host them, fully object-oriented components that work in a more logical manner.

Under .NET, dialogs are essentially Windows Forms. Controls are placed on them in the same way that you would build a form application—with a tool or by coding directly.

Using the Common Dialogs

Common dialogs are simple to use and provide a consistent interaction with the underlying operating system. Dialogs are displayed with the `ShowDialog` function. All common dialogs are

derived from the `System.Windows.Forms.CommonDialog` class. You can use this as a base class for custom dialogs that you might create.

The `FileDialog` Family

The most used of the common dialogs must be the file dialogs. Windows Forms defines a file-open and file-save dialog. To use these in your application, you need to put a member of type `OpenFileDialog` or `SaveFileDialog` in your application. Incidentally, both these dialogs are derived from `System.Windows.Forms.FileDialog`.

Listing 3.2.1 shows a simple application with a file-open dialog.

LISTING 3.2.1 fileopen.cs: OpenFileDialog in Action

```
1: using System;
2: using System.Drawing;
3: using System.ComponentModel;
4: using System.Windows.Forms;
5:
6:
7: namespace Sams
8: {
9:
10: class fileopenapp : System.Windows.Forms.Form
11: {
12:
13:     MainMenuItem m_menu;
14:     MenuItem m_filemenu;
15:
16:     void OnOpenFile(Object sender, EventArgs e)
17:     {
18:         OpenFileDialog dlg=new OpenFileDialog();
19:
20:         if(dlg.ShowDialog() == DialogResult.OK)
21:         {
22:             MessageBox.Show("You selected the file "+dlg.FileName);
23:         }
24:     }
25:
26:     void OnExit(Object sender, EventArgs e)
27:     {
28:         Application.Exit();
29:     }
30:
31:     fileopenapp()
32:     {
```

LISTING 3.2.1 Continued

```
33:         m_menu = new MainMenu();
34:         m_filemenu=new MenuItem("&File");
35:         m_menu.MenuItems.Add(m_filemenu);
36:         MenuItem t;
37:
38:         t=new MenuItem("&Open");
39:         t.Click += new EventHandler(OnOpenFile);
40:         m_filemenu.MenuItems.Add(t);
41:
42:         t=new MenuItem("- ");
43:         m_filemenu.MenuItems.Add(t);
44:
45:         t=new MenuItem("E&xit");
46:         t.Click += new EventHandler(OnExit);
47:         m_filemenu.MenuItems.Add(t);
48:
49:         this.Menu = m_menu;
50:     }
51:
52:     public static void Main()
53:     {
54:         Application.Run(new fileopenapp());
55:     }
56: }
57:
58: }// end of Sams namespace
```

This example is the very simplest use of the dialog. Common dialogs provide much more functionality and are used in more complex ways.

A common dialog control is used differently when the application is constructed by Visual Studio.NET for example. In this instance, the application will instantiate a copy of the dialog when the application is run and attach an event source called `FileOK` to a button click handler in the form that uses the dialog. The delegate for this event handler is a little different from that of menu events. The signature for the `CancelEventHandler` is as follows:

```
void FN(object sender, System.ComponentModel.CancelEventArgs e)
```

Listing 3.2.2 shows the equivalent to this approach in a hand-built form.

LISTING 3.2.2 fileopenevent.cs: Event-driven Common Dialog Response

```
1: using System;
2: using System.Drawing;
```

LISTING 3.2.2 Continued

```
3: using System.ComponentModel;
4: using System.Windows.Forms;
5: using System.IO;
6:
7:
8: namespace Sams
9: {
10:
11: class fileopenapp : System.Windows.Forms.Form
12: {
13:
14:     MainMenu m_menu;
15:     MenuItem m_filemenu;
16:
17:     OpenFileDialog openfiledlg1;
18:
19:     void OnOpenFile(Object sender, EventArgs e)
20:     {
21:         openfiledlg1.Filter = "C# files (*.cs)|"+
22:             "*.cs|Bitmap files (*.bmp)|*.bmp";
23:         openfiledlg1.FilterIndex = 1;
24:         openfiledlg1.ShowDialog();
25:     }
26:
27:     void OnExit(Object sender, EventArgs e)
28:     {
29:         Dispose();
30:     }
31:
32:     fileopenapp()
33:     {
34:         m_menu = new MainMenu();
35:         m_filemenu=new MenuItem("&File");
36:         m_menu.MenuItems.Add(m_filemenu);
37:         MenuItem t;
38:
39:         t=new MenuItem("&Open");
40:         t.Click += new EventHandler(OnOpenFile);
41:         m_filemenu.MenuItems.Add(t);
42:
43:         t=new MenuItem("-");
44:         m_filemenu.MenuItems.Add(t);
45:
46:         t=new MenuItem("E&xit");
```

LISTING 3.2.2 Continued

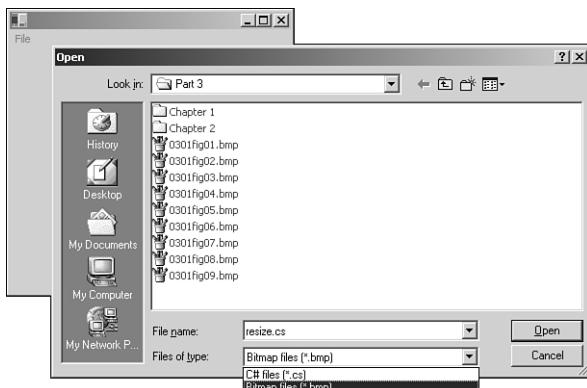
```
47:         t.Click += new EventHandler(OnExit);
48:         m_filemenu.MenuItems.Add(t);
49:
50:         this.Menu = m_menu;
51:
52:         openfiledlg1 = new OpenFileDialog();
53:         openfiledlg1.FileOk += new CancelEventHandler(OnFileOpenOK);
54:     }
55:
56:     void OnFileOpenOK(Object sender, CancelEventArgs e)
57:     {
58:         //MessageBox.Show("You selected the file "+openfiledlg1.FileName);
59:         // remember to add "using System.IO;" to the top of this file.
60:         OpenFileDialog dlg = (OpenFileDialog)sender;
61:
62:         Stream FileStream;
63:         if((FileStream = dlg.OpenFile())!=null)
64:         {
65:             //perform the read from FileStream here...
66:             FileStream.Close(); //and tidy up...
67:         }
68:     }
69:
70:     public static void Main()
71:     {
72:         Application.Run(new fileopenapp());
73:     }
74: }
75:
76: }// end of Sams namespace
```

The event in Listing 3.2.2 will only fire if the OK button has been clicked. There is also a HelpRequested event that will fire if the Help button in the dialog is chosen.

A file dialog can be made to present only a particular set of files by specifying a filter or even to open a file and return a file-stream for you. Lines 19–25 of the listing show how a filter is set for a particular selection of file types.

Lines 56–68 define the event handler that is triggered when a file is selected and opened using the dialog box.

Figure 3.2.1 shows a screen shot of `FileOpenDialog` as defined by this code.

**FIGURE 3.2.1**

The `FileDialog` showing file filtering.

`SaveFileDialog` works in a similar manner because it inherits most of its functionality from `FileDialog`. If you use the `OpenFile` function, a chosen file will be created or opened for writing.

3.2

NOTE

Note that `FileDialog`, `OpenFileDialog`, and `SaveFileDialogs` support properties that allow you to change the way files are displayed, checked, or selected. The .NET SDK help files should be consulted for details

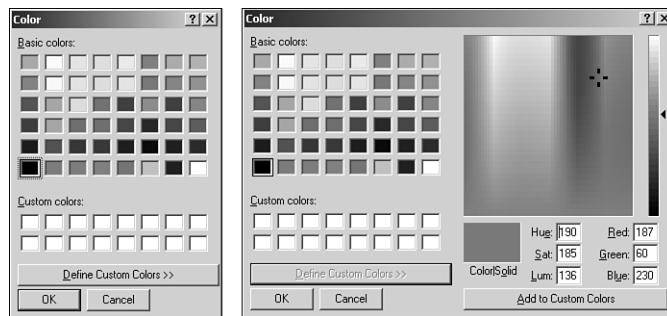
The `ColorDialog`

Selecting color is a common function. Many people try to write color pickers themselves for reasons of aesthetics, but the Windows `ColorDialog` is simple to use and effective. What's more, it's consistent and familiar to most users.

Invoking `ColorDialog` is the same as other common dialogs by virtue of its base class. Create an instance of the class and call the `ShowDialog` method.

Figure 3.2.2 shows the `ColorDialog` in its various modes.

The `ColorDialog` class returns color information in the form of a `Color` object. This contains the alpha, or transparency, and the red, green, and blue values of the color. There are also methods for ascertaining a color's hue, brightness, and saturation values.

**FIGURE 3.2.2**

The `ColorDialog` in all its glory.

There is a property called `AllowFullOpen` in the `ColorDialog` that allows you to decide if the `Define Custom Colors` button is enabled. You can also use the `FullOpen` property to force the complex color picker to appear. The `CustomColors` property is an array of color values that can be set or interrogated and that fill the custom color boxes on the control.

Listing 3.2.3 shows the `ColorDialog` in a selection of modes.

LISTING 3.2.3 colordlgs.cs: Various Incarnations of the `ColorDialog`

```
1: using System;
2: using System.Drawing;
3: using System.ComponentModel;
4: using System.Windows.Forms;
5:
6:
7: namespace Sams {
8:
9:
10: class ColorStretcher : System.Windows.Forms.Form
11: {
12:
13:     void OnFull(Object sender, EventArgs e)
14:     {
15:         ColorDialog dlg=new ColorDialog();
16:         dlg.FullOpen = true;
17:
18:         dlg.ShowDialog();
19:     }
20:
21:     void OnNoCustom(Object sender, EventArgs e)
```

LISTING 3.2.3 Continued

```
22:    {
23:        ColorDialog dlg=new ColorDialog();
24:        dlg.AllowFullOpen = false;
25:
26:        dlg.ShowDialog();
27:    }
28:
29:    void OnNormal(Object sender, EventArgs e)
30:    {
31:        ColorDialog dlg=new ColorDialog();
32:        dlg.Color = Color.PaleGoldenrod;
33:        dlg.ShowDialog();
34:    }
35:
36:    void OnWithColours(Object sender, EventArgs e)
37:    {
38:        ColorDialog dlg=new ColorDialog();
39:        dlg.FullOpen = true;
40:        // this statement defines the first five of the
41:        // custom color settings as ints the data is
42:        // 0xAARRGGBB where AA is alpha, RR is red,
43:        // GG is green and BB is blue expressed as the
44:        // hexadecimal byte equivalent
45:        dlg.CustomColors = new int[5]{0x00ff8040, 0x00c256fe,
46:                                     0x00aa2005, 0x0004f002, 0x002194b5};
47:
48:        dlg.ShowDialog();
49:    }
50:
51:    ColorStretcher()
52:    {
53:        // first a menu for the choices
54:
55:        MainMenu m=new MainMenu();
56:        MenuItem top,temp;
57:
58:        top=new MenuItem("ColorDialog");
59:        m.MenuItems.Add(top);
60:
61:        temp=new MenuItem("Full");
62:        temp.Click+=new EventHandler(OnFull);
63:        top.MenuItems.Add(temp);
64:
65:        temp=new MenuItem("No custom");
```

LISTING 3.2.3 Continued

```
66:         temp.Click+=new EventHandler(OnNoCustom);
67:         top.MenuItems.Add(temp);
68:
69:         temp=new MenuItem("With Colours");
70:         temp.Click+=new EventHandler(OnWithColours);
71:         top.MenuItems.Add(temp);
72:
73:         temp=new MenuItem("Normal");
74:         temp.Click+=new EventHandler(OnNormal);
75:         top.MenuItems.Add(temp);
76:
77:         this.Menu = m;
78:
79:     }
80:
81:     public static void Main()
82:     {
83:         Application.Run(new ColorStretcher());
84:     }
85:
86: }
87: }
```

Notice how the custom colors are defined as integer values in the `OnWithColours` event handler (lines 36–49).

To get the chosen color from the `ColorDialog`, simply read the contents of the `Color` property. Setting this property before invoking the `ShowDialog` method will pre-select a color. This is useful for changing colors that are already set in an application. This is shown in the `OnNormal` handler. To see the color selection, you must click the Define Custom Colors button. The color selected in the palette is the one that is programmed.

FontDialog

The font you choose for text is very important. Times Roman conveys a sense of solidity and reliability. Tahoma shows technical competence, Shotgun proves that you spent too much of your youth before 1975, and nothing says “I love you” better than 72-point Gigi.

The plethora of fonts available to the modern wordsmith is truly phenomenal, so there is nothing more important than a quick, simple method for choosing the face, size, and weight of a font. The standard font dialog is available to .Net programmers through the `FontDialog` class which, like its `FileDialog` and `ColorDialog` cousins, is derived from the `CommonDialog` class.

The technique for invoking the font dialog is the same as we have seen before. The properties of the dialog and the return data is uniquely dedicated to type face selection.

Listing 3.2.4 shows a simple application with a label and a font selection button. The button is wired to an event that invokes the font dialog. The font dialog itself has many options, set through its properties, that allow you to select the operations the dialog can perform. In the following example, we have enabled the Apply button and attached an event that updates the label text each time the Apply button is clicked.

LISTING 3.2.4 fontdlg.cs: Using the Font Selection Dialog

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

namespace Sams {

    class FontPicker : System.Windows.Forms.Form
    {

        Button b;
        Label l;

        void OnApply(Object sender, System.EventArgs e)
        {
            FontDialog dlg = (FontDialog)sender;
            l.Font=dlg.Font;
        }

        void OnClickedb(Object sender,EventArgs e)
        {
            // Create a new Font Dialog
            FontDialog dlg=new FontDialog();

            //Initialize it with the existing font in the label
            dlg.Font = l.Font;

            //Set the property to allow an "Apply button on the form"
            dlg.ShowApply = true;

            //attach an OnApply event handler
            dlg.Apply += new EventHandler(OnApply);

            if(dlg.ShowDialog() != DialogResult.Cancel)
            {
                l.Font = dlg.Font;
            }
        }
    }
}
```

LISTING 3.2.4 Continued

```
public FontPicker()
{
    this.Size=new Size(416,320);

    l=new Label();
    l.Location = new Point(8,8);
    l.Size = new Size(400,200);
    l.Text = "0 1 2 3 4 5 6 7 8 9 \nabcdefghijklmnopqrstuvwxyz"+
    "\nABCDEFGHIJKLMNOPQRSTUVWXYZ";
    l.Font = new Font("Microsoft Sans Serif", 18f);
    this.Controls.Add(l);

    b=new Button();
    b.Text = "Choose Font";
    b.Click += new EventHandler(OnClickedb);
    b.Location = new Point(8,250);
    b.Size = new Size(400,32);
    this.Controls.Add(b);

}

static void Main()
{
    Application.Run(new FontPicker());
}
```

Build this file with the command line:

```
csc /t:winexe fontdlg.cs
```

The Print and Print Preview Dialogs

Rendering a document onscreen or to the printer requires graphics knowledge that we will cover more in depth later in this book, so we won't go into great detail on that subject at this stage. Just to give you a feel for print and print preview. In Listing 3.2.5 we have created a very simple application that enables you to perform print and print preview.

LISTING 3.2.5 printpreview.cs Printing and Print Preview in Action

```
Totally re-written... 1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Drawing.Printing;
6:     using System.Collections;
7:     using System.ComponentModel;
8:     using System.Windows.Forms;
9:
10:
11:    /// <summary>
12:    ///the ppView is a simple control window that uses a common draw
13:    ///method for both painting on screen and printing or print preview
14:    /// </summary>
15:    public class ppView : System.Windows.Forms.Panel
16:    {
17:        private ArrayList points;
18:
19:        Point mousePoint;
20:
21:
22:        void OnClick(Object sender, MouseEventArgs e)
23:        {
24:            if(points == null)
25:            {
26:                points=new ArrayList();
27:            }
28:
29:            points.Add(mousePoint);
30:            Invalidate();
31:        }
32:
33:        void OnMouseMove(Object sender,MouseEventArgs e)
34:        {
35:            mousePoint = new Point(e.X,e.Y);
36:        }
37:
38:        public ppView()
39:        {
40:            this.MouseDown+=new MouseEventHandler(OnClick);
41:            this.MouseMove+=new MouseEventHandler(OnMouseMove);
```

LISTING 3.2.5 Continued

```
42:         this.BackColor=Color.White;
43:     }
44:
45:     private void DrawSmiley(Point pt, Graphics g)
46:     {
47:         g.FillEllipse(Brushes.Black,pt.X-52,pt.Y-52,104,104);
48:         g.FillEllipse(Brushes.Yellow,pt.X-50,pt.Y-50,100,100);
49:         g.FillEllipse(Brushes.Black,pt.X-30,pt.Y-10,60,40);
50:         g.FillEllipse(Brushes.Yellow,pt.X-35,pt.Y-10,70,35);
51:         g.FillEllipse(Brushes.Black,pt.X-25,pt.Y-15,10,10);
52:         g.FillEllipse(Brushes.Black,pt.X+10,pt.Y-15,10,10);
53:     }
54:
55:     private void DoDraw(Graphics g)
56:     {
57:         if(points == null)
58:         {
59:             return;
60:         }
61:         foreach(Point p in points)
62:         {
63:             DrawSmiley(p,g);
64:         }
65:     }
66:
67:     protected override void OnPaint(PaintEventArgs e)
68:     {
69:         if(points == null)
70:         {
71:             return;
72:         }
73:         DoDraw(e.Graphics);
74:     }
75:
76:     public void OnPrintPage(Object sender,PrintPageEventArgs e)
77:     {
78:         if(points == null)
79:         {
80:             return;
81:         }
82:         DoDraw(e.Graphics);
83:     }
84: }
85:
86:
```

LISTING 3.2.5 Continued

```
87:     /// <summary>
88:     /// This is the application that hosts the MainApps.
89:     /// </summary>
90:     public class MDIChildForm : System.Windows.Forms.Form
91:     {
92:         /// <summary>
93:         /// Required designer variable.
94:         /// </summary>
95:         private System.ComponentModel.Container components;
96:         private ppView vw;
97:
98:         // Accessor for private data
99:         public ppView View {
100:             get
101:             {
102:                 return vw;
103:             }
104:         }
105:
106:         public MDIChildForm()
107:         {
108:
109:
110:             //
111:             // Required for Windows Form Designer support
112:             //
113:             InitializeComponent();
114:
115:             vw=new ppView();
116:             vw.Location = new Point(3,3);
117:             vw.Size = new Size(this.Size.Width-6,this.Size.Height-6);
118:             vw.Anchor=AnchorStyles.Left|
119:                 AnchorStyles.Top|
120:                 AnchorStyles.Right|
121:                 AnchorStyles.Bottom;
122:
123:             this.Controls.Add(vw);
124:
125:             //
126:             // TODO: Add any constructor code after InitializeComponent call
127:             //
128:         }
129:
130:         /// <summary>
131:         /// Clean up any resources being used.
```

LISTING 3.2.5 Continued

```
132:         /// </summary>
133:         public override void Dispose()
134:         {
135:             base.Dispose();
136:             components.Dispose();
137:         }
138:
139:         /// <summary>
140:         ///     Required method for Designer support - do not modify
141:         ///     the contents of this method with the code editor.
142:         /// </summary>
143:         private void InitializeComponent()
144:         {
145:             this.components = new System.ComponentModel.Container ();
146:             this.Text = "MDIChildForm";
147:             this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
148:             this.ClientSize = new System.Drawing.Size (328, 277);
149:         }
150:     }
151:
152:     /// <summary>
153:     ///     Summary description for MainApp.
154:     /// </summary>
155:     public class MainApp : System.Windows.Forms.Form
156:     {
157:         /// <summary>
158:         ///     Required designer variable.
159:         /// </summary>
160:         private System.ComponentModel.Container components;
161:         private System.Windows.Forms.MenuItem FileExit;
162:         private System.Windows.Forms.MenuItem FilePrintPreview;
163:         private System.Windows.Forms.MenuItem FilePrint;
164:         private System.Windows.Forms.MenuItem FileNew;
165:         private System.Windows.Forms.MenuItem menuItem1;
166:         private System.Windows.Forms.MainMenu mainMenu1;
167:
168:         public MainApp()
169:         {
170:             //
171:             // Required for Windows Form Designer support
172:             //
173:             InitializeComponent();
174:             //
175:         }
```

LISTING 3.2.5 Continued

```
176:             // Add any constructor code after InitializeComponent call
177:             //
178:         }
179:
180:         /// <summary>
181:         ///     Clean up any resources being used.
182:         /// </summary>
183:         public override void Dispose()
184:         {
185:             base.Dispose();
186:             components.Dispose();
187:         }
188:
189:         /// <summary>
190:         ///     Required method for Designer support - do not modify
191:         ///     the contents of this method with the code editor.
192:         /// </summary>
193:         private void InitializeComponent()
194:         {
195:             this.components = new System.ComponentModel.Container ();
196:             this.mainMenu1 = new System.Windows.Forms.MainMenu ();
197:             this.FilePrintPreview = new System.Windows.Forms.MenuItem ();
198:             this.FileExit = new System.Windows.Forms.MenuItem ();
199:             this.menuItem1 = new System.Windows.Forms.MenuItem ();
200:             this.FileNew = new System.Windows.Forms.MenuItem ();
201:             this.FilePrint = new System.Windows.Forms.MenuItem ();
202:             mainMenu1.MenuItems.AddRange(
203:                 new System.Windows.Forms.MenuItem[1] {this.menuItem1});
204:             FilePrintPreview.Text = "P&review...";
205:             FilePrintPreview.Index = 2;
206:             FilePrintPreview.Click +=
207:                 new System.EventHandler (this.FilePrintPreview_Click);
208:             FileExit.Text = "&Exit";
209:             FileExit.Index = 3;
210:             FileExit.Click += new System.EventHandler (this.FileExit_Click);
211:             menuItem1.Text = "&File";
212:             menuItem1.Index = 0;
213:             menuItem1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[4]
214:                 {this.FileNew,
215:                  this.FilePrint,
216:                  this.FilePrintPreview,
217:                  this.FileExit});
218:             FileNew.Text = "&New";
219:             FileNew.Index = 0;
```

LISTING 3.2.5 Continued

```
220:     FileNew.Click += new System.EventHandler (this.FileNew_Click);
221:     FilePrint.Text = "&Print...";
222:     FilePrint.Index = 1;
223:     FilePrint.Click += new System.EventHandler (this.FilePrint_Click);
224:     this.Text = "MainApp";
225:     this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
226:     this.IsMdiContainer = true;
227:     this.Menu = this.mainMenu1;
228:     this.ClientSize = new System.Drawing.Size (368, 289);
229: }
230:
231: protected void FileExit_Click (object sender, System.EventArgs e)
232: {
233:     Application.Exit();
234: }
235:
236: protected void FilePrintPreview_Click(object sender,
237: =>System.EventArgs e)
237: {
238:     PrintPreviewDialog d=new PrintPreviewDialog();
239:     PrintDocument doc = new PrintDocument();
240:
241:     MDIChildForm f = (MDIChildForm)this.ActiveMdiChild;
242:     ppView vw = f.View;
243:     doc.PrintPage += new PrintPageEventHandler(vw.OnPrintPage);
244:
245:     d.Document=doc;
246:
247:     d.ShowDialog();
248:
249:     doc.PrintPage -= new PrintPageEventHandler(vw.OnPrintPage);
250: }
251:
252: protected void FilePrint_Click (object sender, System.EventArgs e)
253: {
254:     PrintDialog dlg = new PrintDialog();
255:     PrintDocument doc = new PrintDocument();
256:
257:     MDIChildForm f = (MDIChildForm)this.ActiveMdiChild;
258:     ppView vw = f.View;
259:     doc.PrintPage += new PrintPageEventHandler(vw.OnPrintPage);
260:
261:     dlg.Document=doc;
262:
```

LISTING 3.2.5 Continued

```
263:     dlg.ShowDialog();  
264:  
265:     doc.PrintPage -= new PrintPageEventHandler(vw.OnPrintPage);  
266:  
267: }  
268:  
269: protected void FileNew_Click (object sender, System.EventArgs e)  
270: {  
271:     MDIChildForm f = new MDIChildForm();  
272:     f.MdiParent=this;  
273:     f.Show();  
274: }  
275:  
276:     /// <summary>  
277:     /// The main entry point for the application.  
278:     /// </summary>  
279:     public static void Main(string[] args)  
280:     {  
281:         Application.Run(new MainApp());  
282:     }  
283: }  
284: }
```

This application has three components. The form, called `MainApp`, is a Multiple Document Interface (MDI) application and hosts a number of `MDIChildForm` objects. These, in turn, host a simple control called `ppView`.

The `ppView` does all the work of tracking the mouse, adding simple points to an array list, and drawing smiley faces at every stored point. It also employs the technique of a single, simple draw routine that takes a `Graphics` object as an argument.

The print and print preview mechanism uses a `PrintDocument` that fires an event for each printed page. To use the print preview and print facilities, the simple draw routine is used by the `OnPaint`, line 67, and the `OnPrintPage`, line 76, events that call the `DoDraw` method with whatever drawing device context they have been given by the screen or printer drivers. The `Graphics` context passed to the `DoDraw` routine when drawing onscreen is different than when printing. The drawing system knows that it needs to change scales and pixel resolution for you, depending on the device.

If you're interested in the graphics used, lines 47–52 draw the smileys.

Creating Dialogs

Under .NET, the interpretation of what is and is not a dialog is largely dependant on the user interface you choose to use on it. As was mentioned before, a modal dialog is something that requires the user to finish his or her interaction before continuing with the rest of the application. Therefore, dialogs become less and less useful as they get more complex. You must make very careful decisions about what functionality goes into the application's forms and what goes in its dialogs.

Modal and Modeless Dialogs

A dialog can be created in the same way as a form, by dragging tools from a tool palette or by laying out components by hand. The `Form` class has a property called `Modal` that will cause the form to be displayed modally when set to `true`. The `Form` method `ShowDialog` sets this property for you and allows you to display forms in a modal fashion. Listing 3.2.6 shows the same dialog form employed in these two different modes.

LISTING 3.2.6 Modes.cs: Creating Modal and Modeless Dialogs

```
1: using System;
2: using System.ComponentModel;
3: using System.Drawing;
4: using System.Windows.Forms;
5: using System.Collections;
6:
7: class ADialog : Form
8: {
9:
10:    private Button ok,can;
11:
12:    public bool Modeless
13:    {
14:        set
15:        {
16:            if(value)
17:            {
18:                ok.Click += new EventHandler(OnCloseModeless);
19:                can.Click += new EventHandler(OnCloseModeless);
20:            }
21:        }
22:    }
23:
24:    void OnCloseModeless(Object sender, EventArgs e)
25:    {
```

LISTING 3.2.6 Continued

```
26:         this.Close();
27:     }
28:
29:     public ADialog()
30:     {
31:         this.Location = new Point(100,100);
32:         this.Size=new Size(200,100);
33:         this.BorderStyle = FormBorderStyle.FixedDialog;
34:
35:         ok = new Button();
36:         ok.Text = "OK";
37:         ok.DialogResult = DialogResult.OK;
38:         ok.Location = new Point(20,10);
39:         ok.Size = new Size(80,25);
40:         this.Controls.Add(ok);
41:
42:         can = new Button();
43:         can.Text = "Cancel";
44:         can.DialogResult = DialogResult.Cancel;
45:         can.Location = new Point(110,10);
46:         can.Size = new Size(80,25);
47:         this.Controls.Add(can);
48:     }
49: }
50:
51:
52: class AnApp : Form
53: {
54:     void OnModeless(Object sender, EventArgs e)
55:     {
56:         ADIALOG dlg = new ADIALOG();
57:         dlg.Text = "Modeless";
58:         dlg.Modeless = true;
59:         dlg.Show();
60:     }
61:
62:
63:     void OnModal(Object sender, EventArgs e)
64:     {
65:         ADIALOG dlg = new ADIALOG();
66:         dlg.Text = "Modal";
67:         dlg.ShowDialog();
68:     }
69: }
```

LISTING 3.2.6 Continued

```
70:     public AnApp()
71:     {
72:         this.Size=new Size(400,100);
73:         this.FormBorderStyle = FormBorderStyle.FixedDialog;
74:         this.Text = "Dialog Mode Tester";
75:
76:         Button modal = new Button();
77:         modal.Text = "New Modal dialog";
78:         modal.Location = new Point(10,10);
79:         modal.Size = new Size(180,25);
80:         modal.Click += new EventHandler(OnModal);
81:         this.Controls.Add(modal);
82:
83:         Button modeless = new Button();
84:         modeless.Text = "New Modeless dialog";
85:         modeless.Location = new Point(210,10);
86:         modeless.Size = new Size(180,25);
87:         modeless.Click += new EventHandler(OnModeless);
88:         this.Controls.Add(modeless);
89:     }
90:
91:     static void Main()
92:     {
93:         Application.Run( new AnApp());
94:     }
95: }
```

In Listing 3.2.6, the main application is a simple form with two buttons. One button instantiates a new modeless dialog. The other button instantiates a modal dialog.

Lines 70–89 set up the main application window and tie the buttons to the click handlers on lines 52–60 and lines 63–68.

Lines 7–49 define a dialog box that also has two buttons. When running from the `ShowDialog` command, this form will automatically put the dialog return value in the right place and close the form. When running as a modeless dialog, there is an extra setup property on lines 12–22 that ties a click handler to the dialog buttons. Without this handler (defined on lines 24–27), the dialog would not close in response to a button click, only by choosing the Close button on the main window bar.

Notice that you can bring up as many modeless forms as you want, but as soon as a modal dialog is shown, interaction with the main application and all the other modeless forms is blocked.

Closing the main application will close all of the modeless dialogs that are open.

Data Transfer to and from Dialog Members

In its simplest form, a dialog might only transfer data about which button was clicked to exit the dialog. A more complex dialog might make properties available to the programmer that have other information about more complex controls in the dialog. A form run with the `ShowDialog` method is closed by setting the `DialogResult` property of the form. This value is then returned to the code that invoked the dialog.

Listing 3.2.7 shows a simple dialog form with this type of behavior.

LISTING 3.2.7 simpledialog.cs: A simple Dialog

```
1: using System;
2: using System.Drawing;
3: using System.ComponentModel;
4: using System.Windows.Forms;
5:
6: namespace Sams {
7:
8: class SimpleDialog : Form
9: {
10:    public SimpleDialog()
11:    {
12:        // Create two buttons.
13:        Button OkButton=new Button();
14:        OkButton.Text = "Ok";
15:        OkButton.DialogResult = DialogResult.OK;
16:        OkButton.Location = new Point(8,20);
17:        OkButton.Size = new Size(50,24);
18:        this.Controls.Add(OkButton);
19:
20:        Button CancelButton=new Button();
21:        CancelButton.Text = "Cancel";
22:        CancelButton.DialogResult = DialogResult.Cancel;
23:        CancelButton.Location = new Point(64,20);
24:        CancelButton.Size = new Size(50,24);
25:        this.Controls.Add(CancelButton);
26:
27:        this.Text="Dialog";
28:        this.Size = new Size(130,90);
29:        this.FormBorderStyle = FormBorderStyle.FixedDialog;
30:        this.StartPosition = FormStartPosition.CenterParent;
31:        this.ControlBox = false;
```

LISTING 3.2.7 Continued

```
32:     }
33: }
34:
35:
36: class AnApp : Form
37: {
38:     void OnTest(Object sender, EventArgs e)
39:     {
40:         SimpleDialog dlg = new SimpleDialog();
41:         dlg.Owner = this;
42:
43:         if(dlg.ShowDialog() == DialogResult.OK)
44:             MessageBox.Show("You clicked Ok");
45:         else
46:             MessageBox.Show("You clicked Cancel");
47:
48:     }
49:
50:     public AnApp()
51:     {
52:         this.Menu = new MainMenu();
53:         this.Menu.MenuItems.Add(new MenuItem("Dialog"));
54:         this.Menu.MenuItems[0].MenuItems.Add(new MenuItem("Test"));
55:         this.Menu.MenuItems[0].MenuItems[0].Click +=
➥         new EventHandler(OnTest);
56:     }
57:
58:     static void Main()
59:     {
60:         Application.Run(new AnApp());
61:     }
62: }
63: }
```

Dialogs, because they are forms, can host all the Windows Forms controls. Dialogs can be used to return simple information from check boxes, radio buttons, edit controls, and other components. To do this, you need to be able to return more than the simple dialog result seen in the previous example.

Data from dialogs can be accessed via public data members or through methods and properties that you add especially for the task. If you are a veteran MFC programmer, you might want to change your habits from the former method to the latter. Accessor properties play a big role in .NET object interaction.

The interaction between the user and the dialog can be quite complex. A button click, check box selection, or other action may generate events that are trapped by your dialog box code and acted on before the OK button is finally clicked. The design of a dialog can be as exacting as the design of the form that hosts the dialog.

Validation

Some controls, such as the `NumericUpDown` control, have specific behavior for validation. This control has a `Minimum` and `Maximum` property that can be used to define a numeric range for the control. The control will also fire an event when it is validating itself and again when it is validated. You can trap these events, but really the Windows Forms user interface experience is a little rough-and-ready when compared to the ease of using MFC because there is no specific DDX behavior. The validation events are inherited from `System.Windows.Forms.Control` and, in the case of forms shown with the `ShowDialog` method, are generated whenever the control loses focus. This means that validation will take place as you move from control to control within a dialog. This includes clicking a button that sets the `DialogResult` property and closes the dialog. An invalid condition will prevent the dialog box from closing.

Listing 3.2.8 shows a simple application that uses validation to ensure that a numeric value is correct.

3.2

LISTING 3.2.8 dialogValid.cs: Retrieving validated Data from the Dialog

```
1: using System;
2: using System.ComponentModel;
3: using System.Drawing;
4: using System.Windows.Forms;
5: using System.IO;
6: using System.Text;
7:
8: namespace Sams {
9:
10: class DialogValid : System.Windows.Forms.Form
11: {
12:
13:     private Button okButton;
14:     private Button cancelButton;
15:     private NumericUpDown num;
16:
17:     public decimal Num {
18:         get { return num.Value; }
19:         set { num.Value = value; }
20:     }
21: }
```

LISTING 3.2.8 Continued

```
22:     void OnValidating(Object sender, CancelEventArgs e)
23:     {
24:         MessageBox.Show("NumericUpDown is validating");
25:     }
26:
27:     void OnValid(Object sender, EventArgs e)
28:     {
29:         MessageBox.Show("NumericUpDown is valid");
30:     }
31:
32:     public DialogValid()
33:     {
34:
35:         Size = new Size(400,100);
36:         FormBorderStyle = FormBorderStyle.FixedDialog;
37:         Text = "Dialog test";
38:
39:         //place the buttons on the form
40:         okButton = new Button();
41:         okButton.DialogResult = DialogResult.OK;
42:         okButton.Location = new Point(20,28);
43:         okButton.Size = new Size(80,25);
44:         okButton.Text = "OK";
45:         Controls.Add(okButton);
46:
47:         cancelButton = new Button();
48:         cancelButton.Location = new Point(300,28);
49:         cancelButton.Size = new Size(80,25);
50:         cancelButton.Text = "Cancel";
51:         cancelButton.DialogResult = DialogResult.Cancel;
52:         Controls.Add(cancelButton);
53:
54:         // place a label on the form.
55:         Label l = new Label();
56:         l.Text = "NumericUpDown";
57:         l.Location = new Point(20,5);
58:         l.Size = new Size(120,25);
59:         Controls.Add(l);
60:
61:         // finally the numeric control;
62:         num = new NumericUpDown();
63:         num.Location = new Point(140,5);
64:         num.Size = new Size(80,25);
65:         num.Minimum = (decimal)10.0;
```

LISTING 3.2.8 Continued

```
66:         num.Maximum = (decimal)100.0;
67:         num.Value = (decimal)10.0;
68:
69:         //here we add event handlers to show the validating process.
70:         num.Validating+=new CancelEventHandler(OnValidating);
71:         num.Validated+=new EventHandler(OnValid);
72:
73:         Controls.Add(num);
74:
75:     }
76:
77: }
78:
79: class ddApp : System.Windows.Forms.Form
80: {
81:
82:     void OnExit(Object sender, EventArgs e)
83:     {
84:         Application.Exit();
85:     }
86:
87:     void OnDialogTest(Object sender, EventArgs e)
88:     {
89:         DialogValid dlg = new DialogValid();
90:
91:         DialogResult r=dlg.ShowDialog();
92:
93:
94:         StringWriter sw=new StringWriter(new StringBuilder());
95:
96:         sw.WriteLine("Dialog return value = {0}"+
97:         "\nNumericUpDown = {1}",r,dlg.Num);
98:
99:         MessageBox.Show(sw.ToString());
100:
101:    }
102:
103:    public ddApp()
104:    {
105:        MainMenu mm=new MainMenu();
106:        mm.MenuItems.Add(new MenuItem("&Dialog"));
107:        mm.MenuItems[0].MenuItems.Add(new MenuItem("&Test",
108:            new EventHandler(OnDialogTest)));
109:        mm.MenuItems[0].MenuItems.Add(new MenuItem("-"));}
```

LISTING 3.2.8 Continued

```
110:         mm.MenuItems[0].MenuItems.Add(new MenuItem("E&xit",
111:             new EventHandler(OnExit)));
112:         Menu = mm;
113:     }
114:
115:     static void Main()
116:     {
117:         Application.Run(new ddApp());
118:     }
119: }
120:
121: }
```

When you derive your own controls, you can use the `CausesValidation` property to make the `Validating` and `Validated` events fire on your control when the focus changes within the form.

Using Controls

So far, we have seen the `Button` controls and the `NumericUpDown` in action. Let's take a closer look at some of the more commonly used form controls.

Check Boxes and Radio Buttons

Check box and radio button behavior is essentially the same as that of their MFC counterparts. However, the Windows Forms versions do not have any of the nice DDX traits we are all so used to, so reading them and reacting to them is a little more time consuming.

Check boxes can be placed on the form or within other panels or group boxes but, because they don't generally rely on the state of other check boxes around them, their setup is relatively simple.

Radio buttons that are associated with one another should be children of another object, such as a panel or group box, for their automatic radio-button selection to work.

Retrieving the state of these controls is simply a matter of interrogating them directly or using their click methods to keep a record of the user's choices.

Listing 3.2.9 shows a dialog with both check boxes and radio buttons. The code shows several methods for reading the button states or trapping events.

LISTING 3.2.9 dialogtest.cs: Using Check Boxes and Radio Buttons

```
1: using System;
2: using System.ComponentModel;
3: using System.Drawing;
4: using System.Windows.Forms;
5: using System.IO;
6: using System.Text;
7:
8: namespace Sams {
9:
10: class DialogTest : System.Windows.Forms.Form
11: {
12:
13:     private Button okButton;
14:     private Button cancelButton;
15:     private CheckBox checkbox;
16:     private GroupBox radiogroup;
17:     private RadioButton radio1,radio2,radio3;
18:
19:     public int Radio;
20:
21:     public bool Check {
22:         get { return checkbox.Checked; }
23:         set { checkbox.Checked = value; }
24:     }
25:
26:     void OnRadio(Object sender,EventArgs e)
27:     {
28:         int n=0;
29:         foreach(Object o in radiogroup.Controls)
30:         {
31:             if(o is RadioButton)
32:             {
33:                 RadioButton r=(RadioButton)o;
34:                 if(r.Checked)
35:                     Radio=n;
36:                 n++;
37:             }
38:         }
39:     }
40:
41:     public DialogTest()
42:     {
43:
44:         Size = new Size(400,300);
```

LISTING 3.2.9 Continued

```
45:         BorderStyle = FormBorderStyle.FixedSingle;
46:         Text = "Dialog test";
47:
48:         //place the buttons on the form
49:         okButton = new Button();
50:         okButton.DialogResult = DialogResult.OK;
51:         okButton.Location = new Point(20,230);
52:         okButton.Size = new Size(80,25);
53:         okButton.Text = "OK";
54:         Controls.Add(okButton);
55:
56:         cancelButton = new Button();
57:         cancelButton.Location = new Point(300,230);
58:         cancelButton.Size = new Size(80,25);
59:         cancelButton.Text = "Cancel";
60:         cancelButton.DialogResult = DialogResult.Cancel;
61:         Controls.Add(cancelButton);
62:
63:         //place the check box
64:         checkbox = new CheckBox();
65:         checkbox.Location = new Point(20,30);
66:         checkbox.Size = new Size(300,25);
67:         checkbox.Text = "CheckBox";
68:         Controls.Add(checkbox);
69:
70:         //place the radiobuttons
71:         //they need to go into a a group box or a panel...
72:         radiogroup = new GroupBox();
73:         radiogroup.Text = "Radio Buttons";
74:         radiogroup.Location = new Point(10,60);
75:         radiogroup.Size = new Size(380,110);
76:         Controls.Add(radiogroup);
77:
78:         radio1 = new RadioButton();
79:         radio1.Location = new Point(10,15);
80:         // remember this is relative to the group box...
81:         radio1.Size = new Size(360,25);
82:         radio1.Click += new EventHandler(OnRadio);
83:         radio1.Text = "Radio Button #1";
84:         radiogroup.Controls.Add(radio1);
85:
86:
87:         radio2 = new RadioButton();
88:         radio2.Location = new Point(10,40);
```

LISTING 3.2.9 Continued

```
89:         // remember this is relative to the group box...
90:         radio2.Size = new Size(360,25);
91:         radio2.Click += new EventHandler(OnRadio);
92:         radio2.Text = "Radio Button #2";
93:         radiogroup.Controls.Add(radio2);
94:
95:
96:         radio3 = new RadioButton();
97:         radio3.Location = new Point(10,70);
98:         // remember this is relative to the group box...
99:         radio3.Size = new Size(360,25);
100:        radio3.Click += new EventHandler(OnRadio);
101:        radio3.Text = "Radio Button #3";
102:        radiogroup.Controls.Add(radio3);
103:
104:    }
105:
106: }
107:
108: class dtApp : System.Windows.Forms.Form
109: {
110:
111:     void OnExit(Object sender, EventArgs e)
112:     {
113:         Application.Exit();
114:     }
115:
116:     void OnDialogTest(Object sender, EventArgs e)
117:     {
118:         DialogTest dlg = new DialogTest();
119:
120:         DialogResult r=dlg.ShowDialog();
121:
122:
123:         StringWriter sw=new StringWriter(new StringBuilder());
124:
125:         sw.WriteLine("Dialog return value = {0}"+
126:                     "\nRadio Buttons = {1}\nCheck box = {2}",
127:                     r,dlg.Radio,dlg.Check);
128:
129:         MessageBox.Show(sw.ToString());
130:
131:     }
132:
```

3.2

LISTING 3.2.9 Continued

```
133:     public dtApp()
134:     {
135:         MainMenu mm=new MainMenu();
136:         mm.MenuItems.Add(new MenuItem("&Dialog"));
137:         mm.MenuItems[0].MenuItems.Add(new MenuItem("&Test",
138:             new EventHandler(OnDialogTest)));
139:         mm.MenuItems[0].MenuItems.Add(new MenuItem("-"));
140:         mm.MenuItems[0].MenuItems.Add(new MenuItem("E&xit",
141:             new EventHandler(OnExit)));
142:         Menu = mm;
143:     }
144:
145:     static void Main()
146:     {
147:         Application.Run(new dtApp());
148:     }
149: }
150:
151: }
```

Looking more closely at the code in Listing 3.2.9, you can see that the radio buttons are, in fact, children of the group box control (lines 70–103), and that they all use the same click handler (lines 26–39). This radiobutton handler reads through the child controls of the group box one at a time. When it finds a radio button, it looks to see if it's checked. If it is, the corresponding value is set in the forms member data. This is one of many techniques you can use to read radio buttons. You could just as easily define one handler per button and set a specific value for each of them.

In the case of radio buttons, the dialog data is stored in a public data member. This can be read directly by external objects. The check box data is read by a public property that accesses the actual check box control.

Simple Edit Controls

There is a simple text box that handles multiline editing and password display for entering text into your forms. The `TextBox` class can also allow the user to use the Tab key and press Enter so multiline editing is more flexible.

Text edit boxes are illustrated later in the chapter in Listing 3.2.10.

List Boxes

Windows Forms list boxes can be used to display information in vertical or horizontal lists.

List box usage is illustrated in Listing 3.2.10, shown later in this chapter.

Tree Views

The tree view has become a standard for displaying the hierarchical structures of disk directories and other information, such as XML data. Naturally, Windows Forms has a very good tree view class that has great capabilities.

Tree views hold tree nodes. Each node can have a text label associated with it and have other properties such as images that denote the state of the node. Nodes themselves have a `Nodes` collection in which they store their child nodes and so on.

Think of structure of a `TreeView` class as being similar to that of the menu and menu-item relationship.

Tree views support many events for detecting changes in the nodes. The `Beforexxxxx` events will allow you to trap a change before it is effected and cancel it. The `Afterxxxxx` events inform you of the changes that have been made.

Tree views support three event types aside from the ones associated with `RichControl` from which it derives. These event types are `TreeViewEventHandler`, `TreeViewCancelEventHandler`, and `NodeLabelEditEventHandler`. There are also `EventArgs` derivatives to go along with these event types.

The `TreeView` class, some node operations, and the events associated with them are illustrated shortly in Listing 3.2.10.

3.2

USER INTERFACE
COMPONENTS

Tab Controls

Tab controls can be used to create forms with tabbed pages. They consist of a tab bar that can be selected by the user and a collection of `TabPage` objects that can host other form controls. Listing 3.2.10, later in this chapter, shows tab controls and a host of other controls.

Dynamic Control Management

As you have seen from all of the previous demonstrations in this chapter, the addition, placement, and programming of controls is all performed at runtime, with no reference to layout resources. This implies that Windows Forms controls are easier to use when it comes to creating forms and dialogs that morph and reshape themselves in reaction to the user's commands. Probably everyone who is familiar with MFC will have tried, at one time or another, to create a dynamically changing dialog and have been frustrated by the difficulty of adding and removing controls and handling the messages they generate. Listing 3.2.10, later in this chapter, shows an application with all of the elements shown previously plus some dynamic control features.

About the MungoTabApp

The application laid out in Listing 3.2.10 is rather large but illustrates many of the important principles of Windows Forms development. The code listing is numbered but is interspersed with comments in the form of notes in addition to the code comments we have added. For your own sanity, do not try typing this application. It's supplied with this book or available for download from the Sams Publishing Web site. (See the Appendixes in this book for information.)

NOTE

At this point, it is important to note that the application shown in Listing 3.2.10 has nothing like the structure of an application built with VisualStudio.NET. To use such an application for educational purposes is an almost impossible task. The form layout engine puts code into a routine called `InitializeComponent` and does not make any effort to order it in a logical manner. The following application has been laid out as logically as possible for your benefit.

LISTING 3.2.10 MungoTabApp.cs: Tabbed Windows and Controls

```
1: namespace Sams
2: {
3:     using System;
4:     using System.ComponentModel;
5:     using System.Drawing;
6:     using System.Windows.Forms;
7:     using System.Text;
8:
9:
10:    /// <summary>
11:    ///      The MungoTabApp is to show off many of the capabilities and
12:    ///      ease of use of the Windows Forms system. The application
13:    ///      is constructed as a form with a tab control along the bottom.
14:    ///      Each page in the tab control shows off a different aspect of
15:    ///      Windows Forms usage
16:    /// </summary>
17:    public class MungoTabApp : Form
18:    {
```

NOTE

The variables and members in the application are all private. This is good practice for encapsulation. First, all the components are declared.

LISTING 3.2.10 Continued

```
19:          // basic parts
20:          private Timer timer1;
21:          private MainMenu mainMenu1;
22:          private TabControl MainTabControl;
23:          privateTabPage WelcomeTabPage;
24:          privateTabPage SimpleTabPage;
25:          privateTabPage DynamicTabPage;
26:          privateTabPage ListBoxTabPage;
27:          privateTabPage MouseTabPage;
28:          privateTabPage TreeTabPage;
29:
30:
31:
32:          //Welcome page.
33:          private RichTextBox WelcomeTextBox;
34:
35:          //Controls for the Simple Controls page
36:          private Label label1;
37:          private Label label2;
38:          private LinkLabel linkLabel1;
39:          private TextBox ClearTextBox;
40:          private TextBox PasswordTextBox;
41:          private GroupBox groupBox1;
42:          private RadioButton radioButton1;
43:          private RadioButton radioButton2;
44:          private Panel panel1;
45:          private RadioButton radioButton3;
46:          private RadioButton radioButton4;
47:          private Button button1;
48:          private CheckBox checkBox1;
49:          private CheckBox checkBox2;
50:
51:          // the listbox page
52:          private ListBox listBox1;
53:          private CheckedListBox checkedListBox1;
54:          private Label label3;
55:          private Label label4;
56:          private Label PickAWord;
57:          private ComboBox comboBox1;
58:          private ListView listView1;
59:          private DateTimePicker dateTimePicker1;
60:          private Label label16;
61:          private Label label17;
62:          private MonthCalendar monthCalendar1;
```

LISTING 3.2.10 Continued

```
63:     private Label label10;
64:     private TrackBar trackBar1;
65:     private ProgressBar progressBar1;
66:     private Label label8;
67:     private DomainUpDown domainUpDown1;
68:     private NumericUpDown numericUpDown1;
69:     private Label label9;
70:     private Label label11;
71:
72:
73: //Mouse movement and dynamic placement
74: private Button ClickMeButton;
75:
76: // Dynamic controls
77: private CheckBox ShowDynamic;
78: private CheckBox UseAlternates;
79: private CheckBox HideChecks;
80: private GroupBox DynGroup;
81: private RadioButton DynRadioButtn1;
82: private RadioButton DynRadioButtn2;
83: private RadioButton DynRadioButtn3;
84: private RadioButton DynRadioButtn4;
85: private ListBox EventList1;
86: private ListBox EventList2;
87: private Button ClearEvents1;
88: private Button ClearEvents2;
89: //TreeView tab
90: private TreeView treeView1;
91: private ListBox tvlistBox;
92: private Button button4;
93: private Button button5;
94:
95:
96: private bool ShowingRadioGroup;
97:
```

NOTE

The next section initializes the Welcome page. Sections initializing each of the Tab control pages follow.

LISTING 3.2.10 Continued

```

98:         private void InitWelcome()
99:         {
100:             WelcomeTextBox=new RichTextBox();
101:             WelcomeTabPage = new TabPage();
102:             WelcomeTabPage.Text = "Welcome";
103:             WelcomeTabPage.Size = new System.Drawing.Size(576, 422);
104:             WelcomeTabPage.TabIndex = 0;
105:             WelcomeTextBox.Text = "Welcome to the Mungo Tab App.\n"+
106:                 "This Windows Forms demonstration" +
107:                 " application accompanies the
➥Sams C# and the .NET framework"+
108:                 " book by Bob Powell and Richard Weeks.\n\nThis tab hosts a"+
109:                 " RichTextBox. You can edit this text if you wish."+
110:                 "\n\nThe tabs in this form will show you"+
111:                 " some of the more complex controls that
➥ you can use in your "+

112:                 "Windows Forms application.\n\nPlease
➥examine the source code"+
113:                 " for this application carefully.\n\nBob Powell.\n";
114:             WelcomeTextBox.Size = new System.Drawing.Size(576, 424);
115:             WelcomeTextBox.TabIndex = 0;
116:             WelcomeTextBox.Anchor = AnchorStyles.Top |
117:                 AnchorStyles.Left |
118:                 AnchorStyles.Right |
119:                 AnchorStyles.Bottom;
120:             WelcomeTextBox.Visible = true;
121:             WelcomeTabPage.Controls.Add(WelcomeTextBox);
122:             MainTabControl.Controls.Add(WelcomeTabPage);
123:         }
124:

```

NOTE

The handlers dedicated to the simple page are in the next section.

```

125:         // Handlers for the simple page
126:
127:         private void OnClickedSimple1(Object sender, EventArgs e)
128:         {

```

LISTING 3.2.10 Continued

```
129:           // This is one of two handlers that may be attached
➥ to the button
130:           string message = "You clicked the big button";
131:           if(this.checkBox1.Checked)
132:           {
133:               message = "And the password is...."
➥ "+this.PasswordTextBox.Text;
134:           }
135:
136:           MessageBox.Show(message);
137:       }
138:
139:       private void OnCheckColorEdit(Object sender,EventArgs e)
140:       {
141:           // this handler add or removes a second
➥ click handler to the button.
142:
143:           if(this.checkBox2.Checked)
144:           {
145:               this.button1.Click += new EventHandler(OnColorEdit);
146:           }
147:           else
148:           {
149:               this.button1.Click -= new EventHandler(OnColorEdit);
150:           }
151:       }
152:
153:       private void OnColorEdit(Object sender, EventArgs e)
154:       {
155:           // This second handler is added to the
➥ click event of the button
156:           // when the check box is checked.
157:           ColorDialog dlg = new ColorDialog();
158:           if(dlg.ShowDialog() == DialogResult.OK)
159:           {
160:               this.panel1.BackColor=dlg.Color;
161:           }
162:       }
163:
164:       //This handler is invoked by clicking the link label
165:       //it will take you to a web site.
166:       private void LinkClick(Object sender, EventArgs e)
167:       {
168:           this.linkLabel1.LinkVisited=true;
```

LISTING 3.2.10 Continued

```
169:             if(this.ClearTextBox.Text=="This is an editable text box")
170:             {
171:                 System.Diagnostics.Process.Start("IExplore.exe ",
172:                     "http://www.bobpowell.net/");
173:             }
174:             else
175:             {
176:                 try
177:                 {
178:                     System.Diagnostics.Process.Start(ClearTextBox.Text);
179:                 }
180:                 catch(Exception)
181:                 {
182:                     MessageBox.Show("Cannot start
➥process "+ClearTextBox.Text);
183:                 }
184:             }
185:             this.linkLabel1.Text = "Been there, Done that!";
186:         }
187:
188:         //This handler is invoked each time the text in the clear text box
189:         // is modified. It transfers the text to the link button.
190:         // but only if the link has been visited.
191:         private voidTextChanged(Object sender, EventArgs e)
192:         {
193:             if(linkLabel1.LinkVisited )
194:             {
195:                 linkLabel1.Text = ClearTextBox.Text;
196:             }
197:
198:         }
199:
```

Note

The simple controls page is initialized here.

Just before the actual initialization, we have placed the handlers that are associated with the events on this page. This structure is repeated throughout the application.

LISTING 3.2.10 Continued

```
200:      // initializes the simple page.
201:      private void InitSimple()
202:      {
203:          SimpleTabPage = new TabPage();
204:          SimpleTabPage.Size = new System.Drawing.Size(576, 422);
205:          SimpleTabPage.TabIndex = 1;
206:          SimpleTabPage.Text = "Simple controls";
207:
208:          button1 = new Button();
209:          button1.Location = new System.Drawing.Point(32, 240);
210:          button1.Size = new System.Drawing.Size(520, 32);
211:          button1.TabIndex = 7;
212:          button1.Text = "Buttons can be clicked...";
213:          button1.Click+=new EventHandler(OnClickedSimple1);
214:          checkBox1 = new CheckBox();
215:          checkBox1.Location = new System.Drawing.Point(32, 288);
216:          checkBox1.Size = new System.Drawing.Size(520, 16);
217:          checkBox1.TabIndex = 8;
218:          checkBox1.Text =
219:              "Checking this box will make the button "+
220:              "above say whats in the password box";
221:          checkBox2 = new CheckBox();
222:          checkBox2.Location = new System.Drawing.Point(32, 327);
223:          checkBox2.Size = new System.Drawing.Size(520, 16);
224:          checkBox2.TabIndex = 9;
225:          checkBox2.Text = "Checking this box will make the button" +
226:              " above edit the colour of the text panel";
227:          checkBox2.Click += new EventHandler(OnCheckColorEdit);
228:
229:          ClearTextBox = new TextBox();
230:          ClearTextBox.Location = new System.Drawing.Point(344, 8);
231:          ClearTextBox.Size = new System.Drawing.Size(216, 20);
232:          ClearTextBox.TabIndex = 2;
233:          ClearTextBox.Text = "This is an editable text box";
234:
235:          domainUpDown1 = new DomainUpDown();
236:          domainUpDown1.AccessibleName = "DomainUpDown";
237:          domainUpDown1.AccessibleRole = AccessibleRole.ComboBox;
238:          domainUpDown1.Location = new System.Drawing.Point(128, 368);
239:          domainUpDown1.Size = new System.Drawing.Size(144, 20);
240:          domainUpDown1.TabIndex = 10;
241:          domainUpDown1.Text = "domainUpDown1";
242:          domainUpDown1.Items.AddRange(new object []{"England",
243:                                         "Africa",
244:                                         "Mongolia",
245:                                         "Japan"});
```

LISTING 3.2.10 Continued

```
246:         groupBox1 = new GroupBox();
247:         groupBox1.Location = new System.Drawing.Point(8, 80);
248:         groupBox1.Size = new System.Drawing.Size(560, 80);
249:         groupBox1.TabIndex = 5;
250:         groupBox1.TabStop = false;
251:         groupBox1.Text = "A GroupBox";
252:
253:
254:         label1 = new Label();
255:         label1.Location = new System.Drawing.Point(8, 8);
256:         label1.Size = new System.Drawing.Size(144, 24);
257:         label1.TabIndex = 0;
258:         label1.Text = "This is a label control";
259:
260:         label2 = new Label();
261:         label2.Location = new System.Drawing.Point(8, 41);
262:         label2.Size = new System.Drawing.Size(328, 24);
263:         label2.TabIndex = 4;
264:         label2.Text = "The edit box to the right has a
➥password character";
265:
266:         label9 = new Label();
267:         label9.Location = new System.Drawing.Point(16, 368);
268:         label9.Size = new System.Drawing.Size(104, 24);
269:         label9.TabIndex = 12;
270:         label9.Text = "DomainUpDown";
271:
272:         label10 = new Label();
273:         label10.Location = new System.Drawing.Point(276, 370);
274:         label10.Size = new System.Drawing.Size(104, 24);
275:         label10.TabIndex = 13;
276:         label10.Text = "NumericUpDown";
277:
278:         linkLabel1 = new LinkLabel();
279:         linkLabel1.Location = new System.Drawing.Point(152, 8);
280:         linkLabel1.Size = new System.Drawing.Size(176, 24);
281:         linkLabel1.TabIndex = 1;
282:         linkLabel1.TabStop = true;
283:         linkLabel1.Text = "Link labels are like hypertext links";
284:         linkLabel1.Click += new EventHandler(LinkClick);
285:
286:         numericUpDown1 = new NumericUpDown();
287:         numericUpDown1.BeginInit();
288:         numericUpDown1.EndInit();
289:         numericUpDown1.Location = new System.Drawing.Point(392, 368);
290:         numericUpDown1.Size = new System.Drawing.Size(176, 20);
```

LISTING 3.2.10 Continued

```
291:         numericUpDown1.TabIndex = 11;
292:
293:         panel1 = new Panel();
294:         panel1.BackColor =
295:             (System.Drawing.Color)System.Drawing.
➥Color.FromArgb((byte)255,
296:                         (byte)255,
297:                         (byte)128);
298:         panel1.BorderStyle = (BorderStyle)FormBorderStyle.FixedSingle;
299:         panel1.Location = new System.Drawing.Point(8, 168);
300:         panel1.Size = new System.Drawing.Size(560, 64);
301:         panel1.TabIndex = 6;
302:
303:         radioButton1 = new RadioButton();
304:         radioButton1.Location = new System.Drawing.Point(16, 24);
305:         radioButton1.Size = new System.Drawing.Size(504, 16);
306:         radioButton1.TabIndex = 0;
307:         radioButton1.Text = "RadioButtons";
308:
309:         radioButton2 = new RadioButton();
310:         radioButton2.Location = new System.Drawing.Point(16, 48);
311:         radioButton2.Size = new System.Drawing.Size(504, 16);
312:         radioButton2.TabIndex = 1;
313:         radioButton2.Text = "In a groupBox are used to isolate";
314:
315:         radioButton3 = new RadioButton();
316:         radioButton3.Location = new System.Drawing.Point(16, 8);
317:         radioButton3.Size = new System.Drawing.Size(536, 16);
318:         radioButton3.TabIndex = 0;
319:         radioButton3.Text = "Other radio buttons";
320:
321:         radioButton4 = new RadioButton();
322:         radioButton4.Location = new System.Drawing.Point(16, 32);
323:         radioButton4.Size = new System.Drawing.Size(536, 16);
324:         radioButton4.TabIndex = 1;
325:         radioButton4.Text = "in other GroupBoxes,
➥or in this case, Panels.";
326:
327:         panel1.Controls.Add(radioButton3);
328:         panel1.Controls.Add(radioButton4);
329:
330:         groupBox1.Controls.Add(radioButton1);
331:         groupBox1.Controls.Add(radioButton2);
332:
```

LISTING 3.2.10 Continued

```
333:         PasswordTextBox = new TextBox();
334:         PasswordTextBox.Location = new System.Drawing.Point(344, 40);
335:         PasswordTextBox.PasswordChar = '*';
336:         PasswordTextBox.Size = new System.Drawing.Size(216, 20);
337:         PasswordTextBox.TabIndex = 3;
338:         PasswordTextBox.Text = "Password";
339:
340:         ClearTextBox.TextChanged += new EventHandler(TextChanged);
341:
342:         SimpleTabPage.Controls.Add(button1);
343:         SimpleTabPage.Controls.Add(checkBox1);
344:         SimpleTabPage.Controls.Add(checkBox2);
345:         SimpleTabPage.Controls.Add(ClearTextBox);
346:         SimpleTabPage.Controls.Add(domainUpDown1);
347:         SimpleTabPage.Controls.Add(groupBox1);
348:         SimpleTabPage.Controls.Add(label1);
349:         SimpleTabPage.Controls.Add(label10);
350:         SimpleTabPage.Controls.Add(label12);
351:         SimpleTabPage.Controls.Add(label19);
352:         SimpleTabPage.Controls.Add(linkLabel1);
353:         SimpleTabPage.Controls.Add(numericUpDown1);
354:         SimpleTabPage.Controls.Add(panel11);
355:         SimpleTabPage.Controls.Add(PasswordTextBox);
356:     }
357:
```

Note

Event handlers for the list box tab are in the following section.

3.2

```
358:         // List box tab event handlers.
359:         // This handler transfers the value of the trackbar
➥to the progress bar.
360:         private void OnTrack(Object sender, EventArgs e)
361:         {
362:             TrackBar b=(TrackBar)sender;
363:             this.progressBar1.Value = b.Value;
364:         }
365:
366:         // This handler constructs a sentence from the
➥checked items in the list
367:         // and displays it in a label to the right of the control.
```

LISTING 3.2.10 Continued

```
368:         private void CheckedListHandler(Object sender,
➥ItemCheckEventArgs e)
369:         {
370:             StringBuilder sb=new StringBuilder();
371:             int ni=-1;
372:             if(e.NewValue==CheckState.Checked)
373:                 ni=e.Index;
374:                 for(int i=0;i<checkedListBox1.Items.Count;i++)
375:                 {
376:                     if(i==ni || (i!=e.Index && checkedListBox1.
➥GetItemChecked(i)))
377:                         sb.Append(checkedListBox1.Items[i].ToString()+" ");
378:                     }
379:                     PickAWord.Text = sb.ToString();
380:                 }
381:
382:             // this handler gets the items from the list box
➥and changes their case
383:             // as the mouse passes over them.
384:             private void ListBoxMouseOver(Object sender, MouseEventArgs e)
385:             {
386:                 string s;
387:                 int i=0;
388:                 // first we reset the case of all the strings
389:                 foreach(object o in listBox1.Items)
390:                 {
391:                     s=(string)o;
392:                     listBox1.Items[i++]=s.ToLower();
393:                 }
394:                 i = listBox1.IndexFromPoint(e.X,e.Y);
395:                 if(i>-1)
396:                 {
397:                     s=(string)listBox1.Items[i];
398:                     listBox1.Items[i]=s.ToUpper();
399:                 }
400:             }
401:
402:             // Right clicking the combo box invokes this handler
403:             // it sorts the contents of the dropdown.
404:             private void SortComboboxHandler(Object sender, EventArgs e)
405:             {
406:                 this.comboBox1.Sorted=true;
407:             }
408:
```

LISTING 3.2.10 Continued**NOTE**

The next section illustrates how the list box classes are used. There is also a track bar control and a progress bar with which you can experiment.

```
409:         // List box tab initialization.  
410:         private void InitLists()  
411:         {  
412:             ListBoxTabPage = new TabPage();  
413:             ListBoxTabPage.Size = new System.Drawing.Size(576, 422);  
414:             ListBoxTabPage.TabIndex = 2;  
415:             ListBoxTabPage.Text = "List boxes";  
416:  
417:             checkedListBox1 = new CheckedListBox();  
418:             checkedListBox1.Items.AddRange(new object[] {"The",  
419:                                         "These", "All", "Words", "Men", "Are", "Can", "Be",  
420:                                         "Might", "Not", "Be", "Made", "As", "Happy", "Equal",  
421:                                         "Stupid", "Lost"});  
422:             checkedListBox1.Location = new System.Drawing.Point(216, 8);  
423:             checkedListBox1.Size = new System.Drawing.Size(192, 94);  
424:             checkedListBox1.TabIndex = 1;  
425:             checkedListBox1.CheckOnClick=true;  
426:             checkedListBox1.ItemCheck +=  
427:                 new ItemCheckEventHandler(CheckedListHandler);  
428:  
429:             comboBox1 = new ComboBox();  
430:             comboBox1.Items.AddRange(new object[] {"A",  
431:                                         "Little", "aardvark", "Never", "Hurt",  
432:                                         "Anyone", "5 ", "9 ", "7 ", "1", "0 ",  
433:                                         "2", "4", "3", "6", "8"});  
434:             comboBox1.Location = new System.Drawing.Point(8, 144);  
435:             comboBox1.Size = new System.Drawing.Size(184, 21);  
436:             comboBox1.TabIndex = 5;  
437:             comboBox1.Text = "Context menu sorts";  
438:             ContextMenu m=new ContextMenu();  
439:             MenuItem t=new MenuItem("Sort",new EventHandler  
➥(SortComboboxHandler));  
440:             m.MenuItems.Add(t);  
441:             comboBox1.ContextMenu = m;  
442:  
443:             dateTimePicker1 = new DateTimePicker();  
444:             dateTimePicker1.Location = new System.Drawing.Point(216, 272);
```

LISTING 3.2.10 Continued

```
445:         datePicker1.Size = new System.Drawing.Size(344, 20);
446:         datePicker1.TabIndex = 7;
447:
448:         label3 = new Label();
449:         label3.Location = new System.Drawing.Point(8, 112);
450:         label3.Size = new System.Drawing.Size(184, 16);
451:         label3.TabIndex = 2;
452:         label3.Text = "A Simple list box";
453:         label4 = new Label();
454:         label4.Location = new System.Drawing.Point(224, 112);
455:         label4.Size = new System.Drawing.Size(184, 16);
456:         label4.TabIndex = 3;
457:         label4.Text = "A Checked list box";
458:         label6 = new Label();
459:         label6.Location = new System.Drawing.Point(216, 248);
460:         label6.Size = new System.Drawing.Size(184, 16);
461:         label6.TabIndex = 8;
462:         label6.Text = "A list view";
463:         label7 = new Label();
464:         label7.Location = new System.Drawing.Point(214, 303);
465:         label7.Size = new System.Drawing.Size(184, 16);
466:         label7.TabIndex = 9;
467:         label7.Text = "A DateTimePicker";
468:         label8 = new Label();
469:         label8.Location = new System.Drawing.Point(7, 341);
470:         label8.Size = new System.Drawing.Size(184, 16);
471:         label8.TabIndex = 11;
472:         label8.Text = "The MonthCalender control";
473:
474:         label11 = new Label();
475:         label11.Location = new System.Drawing.Point(7, 384);
476:         label11.Size = new System.Drawing.Size(184, 16);
477:         label11.TabIndex = 14;
478:         label11.Text = "Trackbar and progress bar (Right)";
479:
480:         listBox1 = new ListBox();
481:         listBox1.Items.AddRange( new object[] {"Fish",
482:                                     "Chips", "Vinegar", "Marmite", "Cream Crackers",
483:                                     "Marmalade", "Stilton", "Mushy Peas",
484:                                     "Sherbert Lemons", "Wellie boots", "Spanners"});
485:         listBox1.Location = new System.Drawing.Point(8, 8);
486:         listBox1.Size = new System.Drawing.Size(184, 95);
487:         listBox1.TabIndex = 0;
488:         listBox1.MouseMove += new MouseEventHandler(ListBoxMouseOver);
489:
```

LISTING 3.2.10 Continued

```
490:         listView1 = new ListView();
491:         listView1.ForeColor = System.Drawing.SystemColors.WindowText;
492:         listView1.Items.Add(new ListViewItem("knives"));
493:         listView1.Items.Add(new ListViewItem("forks"));
494:         listView1.Items.Add(new ListViewItem("spoons"));
495:         listView1.Items.Add(new ListViewItem("dogs"));
496:         listView1.Items.Add(new ListViewItem("fish"));
497:         listView1.Location = new System.Drawing.Point(216, 136);
498:         listView1.Size = new System.Drawing.Size(352, 96);
499:         listView1.TabIndex = 6;
500:
501:         monthCalendar1 = new MonthCalendar();
502:         monthCalendar1.Location = new System.Drawing.Point(8, 184);
503:         monthCalendar1.TabIndex = 10;
504:         monthCalendar1.TabStop = true;
505:
506:         progressBar1 = new ProgressBar();
507:         progressBar1.Location = new System.Drawing.Point(216, 344);
508:         progressBar1.Size = new System.Drawing.Size(336, 24);
509:         progressBar1.TabIndex = 13;
510:
511:         PickAWord = new Label();
512:         PickAWord.Location = new System.Drawing.Point(416, 8);
513:         PickAWord.Size = new System.Drawing.Size(152, 96);
514:         PickAWord.TabIndex = 4;
515:
516:         trackBar1 = new TrackBar();
517:         trackBar1.BeginInit();
518:         trackBar1.Location = new System.Drawing.Point(272, 376);
519:         trackBar1.Maximum = 100;
520:         trackBar1.Size = new System.Drawing.Size(184, 42);
521:         trackBar1.TabIndex = 12;
522:         trackBar1.ValueChanged += new EventHandler(OnTrack);
523:         trackBar1.EndInit();
524:
525:
526:
527:         tabPage1.Controls.Add(checkedListBox1);
528:         tabPage1.Controls.Add(comboBox1);
529:         tabPage1.Controls.Add(dateTimePicker1);
530:         tabPage1.Controls.Add(label11);
531:         tabPage1.Controls.Add(label3);
532:         tabPage1.Controls.Add(label4);
533:         tabPage1.Controls.Add(label6);
```

LISTING 3.2.10 Continued

```
534:         ListBoxTabPage.Controls.Add(label17);
535:         ListBoxTabPage.Controls.Add(label18);
536:         ListBoxTabPage.Controls.Add(listBox1);
537:         ListBoxTabPage.Controls.Add(listView1);
538:         ListBoxTabPage.Controls.Add(monthCalendar1);
539:         ListBoxTabPage.Controls.Add(PickAWord);
540:         ListBoxTabPage.Controls.Add(progressBar1);
541:         ListBoxTabPage.Controls.Add(trackBar1);
542:     }
```

NOTE

The event handlers for the Dynamic RadioButtons tab are a little complex. A group of buttons is added or destroyed as needed, and a set of events are added or re-directed according to the user's choice.

```
543:
544: // This is the first of two possible events fired by the
545: // dynamic radio buttons. It adds to a list box.
546: private void RadioEvent1(Object sender, EventArgs e)
547:
548:     RadioButton r = (RadioButton)sender;
549:     this.EventList1.Items.Add("Event #1 from: "+r.Text);
550:
551:
552: // This is the second of two possible events fired by the
553: // dynamic radio buttons. It adds to a list box.
554: private void RadioEvent2(Object sender, EventArgs e)
555:
556:     RadioButton r = (RadioButton)sender;
557:     this.EventList2.Items.Add("Event #2 from: "+r.Text);
558:
559:
560: // This handler clears the first list box out
561: private void Clear1(Object sender,EventArgs e)
562:
563:     this.EventList1.Items.Clear();
564:
565:
566: // This handler clears the second list box out
567: private void Clear2(Object sender,EventArgs e)
```

LISTING 3.2.10 Continued

```
568:         {
569:             this.EventList2.Items.Clear();
570:         }
571:
572:         // This routine removes all events from all radio buttons
573:         // in the dynamic page.
574:         private void RemoveEvents()
575:         {
576:             DynRadioButtn4.Click -= new EventHandler(RadioEvent1);
577:             DynRadioButtn3.Click -= new EventHandler(RadioEvent1);
578:             DynRadioButtn2.Click -= new EventHandler(RadioEvent1);
579:             DynRadioButtn1.Click -= new EventHandler(RadioEvent1);
580:             DynRadioButtn4.Click -= new EventHandler(RadioEvent2);
581:             DynRadioButtn3.Click -= new EventHandler(RadioEvent2);
582:             DynRadioButtn2.Click -= new EventHandler(RadioEvent2);
583:             DynRadioButtn1.Click -= new EventHandler(RadioEvent2);
584:         }
585:
586:         // This method add the correct event handler alternative
587:         // to the radiobuttons on the dynamic page
588:         private void AddEvents()
589:         {
590:
591:             if(!this.UseAlternates.Checked )
592:             {
593:                 DynRadioButtn4.Click += new EventHandler(RadioEvent1);
594:                 DynRadioButtn3.Click += new EventHandler(RadioEvent1);
595:                 DynRadioButtn2.Click += new EventHandler(RadioEvent1);
596:                 DynRadioButtn1.Click += new EventHandler(RadioEvent1);
597:             }
598:             else
599:             {
600:                 DynRadioButtn4.Click += new EventHandler(RadioEvent2);
601:                 DynRadioButtn3.Click += new EventHandler(RadioEvent2);
602:                 DynRadioButtn2.Click += new EventHandler(RadioEvent2);
603:                 DynRadioButtn1.Click += new EventHandler(RadioEvent2);
604:             }
605:         }
606:
607:         // This event handler swaps the dynamic radio
➥button event handlers
608:         public void OnUseAlternates(Object sender, EventArgs e)
609:         {
610:             if(ShowingRadioGroup)
```

LISTING 3.2.10 Continued

```
611:         {
612:             RemoveEvents();
613:             AddEvents();
614:         }
615:     }
616:
617:     // This method removes the whole dynamic radiobutton
618:     // panel, clears the event list and destroys the items
619:     public void RemoveRadio()
620:     {
621:         if(ShowingRadioGroup)
622:         {
623:             DynGroup.Controls.Remove(DynRadioButtn4);
624:             DynGroup.Controls.Remove(DynRadioButtn3);
625:             DynGroup.Controls.Remove(DynRadioButtn2);
626:             DynGroup.Controls.Remove(DynRadioButtn1);
627:             DynamicTabPage.Controls.Remove(DynGroup);
628:
629:             RemoveEvents();
630:             DynamicTabPage.Controls.Remove(DynGroup);
631:
632:             DynRadioButtn4.Dispose();
633:             DynRadioButtn3.Dispose();
634:             DynRadioButtn2.Dispose();
635:             DynRadioButtn1.Dispose();
636:             DynGroup.Dispose();
637:
638:             ShowingRadioGroup = false;
639:
640:         }
641:     }
642:
643:     //This method adds the dynamic radio button group and
644:     //wires up the event handlers.
645:     private void AddRadio()
646:     {
647:         if(!ShowingRadioGroup)
648:         {
649:             DynGroup = new GroupBox();
650:             DynGroup.Location = new System.Drawing.Point(240, 16);
651:             DynGroup.Size = new System.Drawing.Size(312, 120);
652:             DynGroup.TabIndex = 3;
653:             DynGroup.TabStop = false;
654:             DynGroup.Text = "Dynamic radiobuttons";
655:         }
```

LISTING 3.2.10 Continued

```
656:             DynRadioButtn1 = new RadioButton();
657:             DynRadioButtn1.Location = new
➥System.Drawing.Point(8, 24);
658:             DynRadioButtn1.Size = new System.Drawing.Size(296, 16);
659:             DynRadioButtn1.TabIndex = 0;
660:             DynRadioButtn1.Text = "Choice 1";
661:
662:             DynRadioButtn2 = new RadioButton();
663:             DynRadioButtn2.Location = new
➥System.Drawing.Point(8, 41);
664:             DynRadioButtn2.Size = new System.Drawing.Size(296, 16);
665:             DynRadioButtn2.TabIndex = 1;
666:             DynRadioButtn2.Text = "Choice 2";
667:
668:             DynRadioButtn3 = new RadioButton();
669:             DynRadioButtn3.Location =
➥new System.Drawing.Point(8, 64);
670:             DynRadioButtn3.Size = new System.Drawing.Size(296, 16);
671:             DynRadioButtn3.TabIndex = 2;
672:             DynRadioButtn3.Text = "Choice 3";
673:
674:             DynRadioButtn4 = new RadioButton();
675:             DynRadioButtn4.Location =
➥new System.Drawing.Point(8, 88);
676:             DynRadioButtn4.Size = new System.Drawing.Size(296, 16);
677:             DynRadioButtn4.TabIndex = 3;
678:             DynRadioButtn4.Text = "Choice 4";
679:
680:             AddEvents();
681:
682:             DynGroup.Controls.Add(DynRadioButtn4);
683:             DynGroup.Controls.Add(DynRadioButtn3);
684:             DynGroup.Controls.Add(DynRadioButtn2);
685:             DynGroup.Controls.Add(DynRadioButtn1);
686:             DynamicTabPage.Controls.Add(DynGroup);
687:
688:             ShowingRadioGroup = true;
689:         }
690:     }
691:
692:     //This event handler uses helper methods to manage
➥the presence of the
693:     //dynamic radiobutton group and the handlers that they use
694:     private void ShowDynamicEvent(Object sender, EventArgs e)
```

LISTING 3.2.10 Continued

```
695:         {
696:             CheckBox b=(CheckBox) sender;
697:
698:             if(b.Checked)
699:             {
700:                 AddRadio();
701:             }
702:             else
703:             {
704:                 RemoveRadio();
705:             }
706:
707:         }
708:
709:         // This event handler adds or removes the dynamic
➥check buttons and
710:         // repositions the control to make it look neat and tidy.
711:         private void AddChecks(Object sender, EventArgs e)
712:         {
713:             CheckBox c=(CheckBox)sender;
714:
715:             if(this.HideChecks.Checked)
716:             {
717:                 RemoveRadio();
718:                 c.Location = new Point(8,16);
719:                 DynamicTabPage.Controls.Remove(UseAlternates);
720:                 DynamicTabPage.Controls.Remove>ShowDynamic());
721:                 ShowDynamic.Click -= new EventHandler>ShowDynamicEvent);
722:                 UseAlternates.Dispose();
723:                 ShowDynamic.Dispose();
724:             }
725:             else
726:             {
727:                 c.Location = new Point(8,64);
728:
729:                 ShowDynamic = new CheckBox();
730:                 ShowDynamic.Location = new System.Drawing.Point(8, 16);
731:                 ShowDynamic.Size = new System.Drawing.Size(168, 16);
732:                 ShowDynamic.TabIndex = 0;
733:                 ShowDynamic.Text = "Show dynamic RadioButtons";
734:                 ShowDynamic.Click += new EventHandler>ShowDynamicEvent);
735:
736:                 UseAlternates = new CheckBox();
737:                 UseAlternates.Location = new System.Drawing.Point(8, 40);
```

LISTING 3.2.10 Continued

```
738:             UseAlternates.Size = new System.Drawing.Size(168, 16);
739:             UseAlternates.TabIndex = 1;
740:             UseAlternates.Text = "Use alternate handlers";
741:             UseAlternates.Click += new EventHandler(OnUseAlternates);
742:
743:             DynamicTabPage.Controls.Add(UseAlternates);
744:             DynamicTabPage.Controls.Add>ShowDynamic);
745:         }
746:     }
747:
```

Note

Initialization for the Dynamic Controls tab follows in the next section.

```
748: // This method initializes the dynamic buttons tab
749: private void InitDynamic()
750: {
751:
752:     DynamicTabPage = new TabPage();
753:     DynamicTabPage.Size = new System.Drawing.Size(576, 422);
754:     DynamicTabPage.TabIndex = 3;
755:     DynamicTabPage.Text = "Dynamic controls";
756:
757:     ClearEvents1 = new Button();
758:     ClearEvents1.Location = new System.Drawing.Point(48, 328);
759:     ClearEvents1.Size = new System.Drawing.Size(128, 24);
760:     ClearEvents1.TabIndex = 6;
761:     ClearEvents1.Text = "Clear the events";
762:     ClearEvents1.Click += new EventHandler(Clear1);
763:
764:     ClearEvents2 = new Button();
765:     ClearEvents2.Location = new System.Drawing.Point(340, 330);
766:     ClearEvents2.Size = new System.Drawing.Size(128, 24);
767:     ClearEvents2.TabIndex = 7;
768:     ClearEvents2.Text = "Clear the events";
769:     ClearEvents2.Click += new EventHandler(Clear2);
770:
771:     EventList1 = new ListBox();
772:     EventList1.Location = new System.Drawing.Point(16, 176);
773:     EventList1.Size = new System.Drawing.Size(200, 121);
774:     EventList1.TabIndex = 4;
```

LISTING 3.2.10 Continued

```
775:           EventList2 = new ListBox();
776:           EventList2.Location = new System.Drawing.Point(308, 180);
777:           EventList2.Size = new System.Drawing.Size(200, 121);
778:           EventList2.TabIndex = 5;
779:
780:           HideChecks = new CheckBox();
781:           HideChecks.Location = new System.Drawing.Point(8, 64);
782:           HideChecks.Size = new System.Drawing.Size(168, 16);
783:           HideChecks.TabIndex = 2;
784:           HideChecks.Text = "Hide checkboxes";
785:           HideChecks.Click += new EventHandler(AddChecks);
786:           AddChecks(HideChecks,new EventArgs());
787:
788:           DynamicTabPage.Controls.Add(ClearEvents1);
789:           DynamicTabPage.Controls.Add(ClearEvents2);
790:           DynamicTabPage.Controls.Add(EventList2);
791:           DynamicTabPage.Controls.Add(EventList1);
792:           DynamicTabPage.Controls.Add(HideChecks);
793:       }
794:
```

NOTE

Now come the handlers for the mouse interaction tab.

```
795:           //
796:           // Handlers for the mouse tab
797:           //
798:
799:
800:           // This handler moves the button to the
→opposite side of the tab-page
801:           // from the mouse cursor
802:           private void OnMouseMoved(Object sender, MouseEventArgs e)
803:           {
804:               Point center =
805:                   new Point(MouseTabPage.Width/2,MouseTabPage.Height/2);
806:               ClickMeButton.Location =
807:                   new Point(center.X-
→(ClickMeButton.Size.Width/2)-(e.X-center.X),
808:                           center.Y-(ClickMeButton.Size.Height/2)-(e.Y-center.Y));
809:           }
```

LISTING 3.2.10 Continued

```
810:  
811:    //This handler shows when the button is caught and clicked.  
812:    private void OnClickedClickme(Object sender, EventArgs e)  
813:    {  
814:        MessageBox.Show("Caught me!!");  
815:    }  
816:
```

Note

The Mouse Interaction tab initialization is in the following section.

```
817:    // This method initializes the mouse page.  
818:    private void InitMouse()  
819:    {  
820:        MouseTabPage = new TabPage();  
821:        MouseTabPage.Controls.Add(ClickMeButton);  
822:        MouseTabPage.Size = new System.Drawing.Size(576, 422);  
823:        MouseTabPage.TabIndex = 4;  
824:        MouseTabPage.Text = "Mouse interaction";  
825:  
826:        ClickMeButton = new Button();  
827:        ClickMeButton.Location = new System.Drawing.Point(200, 128);  
828:        ClickMeButton.Size = new System.Drawing.Size(184, 112);  
829:        ClickMeButton.TabIndex = 0;  
830:        ClickMeButton.Text = "Click me!";  
831:        ClickMeButton.Click += new EventHandler(OnClickedClickme);  
832:  
833:        MouseTabPage.Controls.Add(ClickMeButton);  
834:  
835:        MouseTabPage.MouseMove +=  
►new MouseEventHandler(OnMouseMoved);  
836:    }  
837:
```

3.2**Note**

The handlers for the tree tab follow in this section. Note how the Beforexxx and Afterxxx action handlers are used to create the list of actions taking place.

LISTING 3.2.10 Continued

```
838:           //Handlers for the tree tab....  
839:  
840:  
841:           //The overloaded list function shows the treee  
➥view events in a list  
842:  
843:           private void List(string s, TreeNode n)  
844:           {  
845:               string o=s+" "+n.Text;  
846:               tvlistBox.Items.Add(o);  
847:           }  
848:  
849:           private void List(string s, string l, TreeNode n)  
850:           {  
851:               string o=s+" (new = "+l+")" current = "+n.Text;  
852:               tvlistBox.Items.Add(o);  
853:           }  
854:  
855:           // These handlers simply reflect the event  
➥type and a little bit of  
856:           // node data to the list box on the right of  
➥the treeView1 control.  
857:           private void OnAfterCheck(Object sender,TreeViewEventArgs e)  
858:           {  
859:               List("AfterCheck", e.Node);  
860:           }  
861:  
862:           private void OnAfterCollapse(Object sender,TreeViewEventArgs e)  
863:           {  
864:               List("AfterCollapse", e.Node);  
865:           }  
866:  
867:           private void OnAfterExpand(Object sender,TreeViewEventArgs e)  
868:           {  
869:               List("AfterExpand", e.Node);  
870:           }  
871:  
872:           private void OnAfterSelect(Object sender,TreeViewEventArgs e)  
873:           {  
874:               List("AfterSelect", e.Node);  
875:           }  
876:           private void OnBeforeCheck(Object  
➥sender,TreeViewCancelEventArgs e)  
877:           {
```

LISTING 3.2.10 Continued

```
878:             List("AfterCollapse", e.Node);
879:         }
880:         private void OnBeforeCollapse(Object
➥sender,TreeViewCancelEventArgs e)
881:         {
882:             List("BeforeCollapse", e.Node);
883:         }
884:         private void OnBeforeExpand(Object
➥sender,TreeViewCancelEventArgs e)
885:         {
886:             List("BeforeExpand", e.Node);
887:         }
888:         private void OnBeforeLabelEdit
➥(Object sender,NodeLabelEditEventArgs e)
889:         {
890:             List("BeforeEdit", e.Label, e.Node);
891:         }
892:
893:         private void OnAfterLabelEdit(Object
➥sender,NodeLabelEditEventArgs e)
894:         {
895:             List("AfterEdit", e.Label, e.Node);
896:         }
897:
898:
899:         private void OnBeforeSelect(Object
➥sender,TreeViewCancelEventArgs e)
900:         {
901:             List("BeforeSelect", e.Node);
902:         }
903:
904:         private void OnAddRoot(Object sender, EventArgs e)
905:         {
906:             button5.Enabled=true;
907:
908:             if(treeView1.Nodes.Count==0)
909:             {
910:                 treeView1.Nodes.Add(new TreeNode("Root node"));
911:             }
912:             else
913:             {
914:                 treeView1.Nodes.Add(new TreeNode("Sibling node"));
915:             }
916:         }
```

LISTING 3.2.10 Continued

```
917:  
918:    private void OnAddChild(Object sender, EventArgs e)  
919:    {  
920:        if(treeView1.SelectedNode==null)  
921:            return;  
922:        treeView1.SelectedNode.Nodes.Add(new TreeNode("Child"));  
923:    }  
924:
```

NOTE

The following section initializes the tree control test tab.

```
925:    //Initializes the tree control.  
926:    private void InitTree()  
927:    {  
928:        TreeTabPage = new TabPage();  
929:        TreeTabPage.Text = "TreeView";  
930:        TreeTabPage.Size = new System.Drawing.Size(576, 422);  
931:        TreeTabPage.TabIndex = 5;  
932:  
933:        treeView1 = new TreeView();  
934:        treeView1.Anchor = AnchorStyles.Left |  
935:            AnchorStyles.Top |  
936:            AnchorStyles.Right |  
937:            AnchorStyles.Bottom;  
938:        treeView1.Size = new System.Drawing.Size(264, 360);  
939:        treeView1.TabIndex = 0;  
940:        treeView1.ShowLines=true;  
941:        treeView1.ShowPlusMinus=true;  
942:        treeView1.ShowRootLines=true;  
943:        treeView1.LabelEdit=true;  
944:  
945:        treeView1.AfterCheck +=  
946:            new TreeViewEventHandler(OnAfterCheck);  
947:        treeView1.AfterCollapse +=  
948:            new TreeViewEventHandler(OnAfterCollapse);  
949:        treeView1.AfterExpand +=  
950:            new TreeViewEventHandler(OnAfterExpand);  
951:        treeView1.AfterSelect +=  
952:            new TreeViewEventHandler(OnAfterSelect);
```

LISTING 3.2.10 Continued

```
953:         treeView1.AfterLabelEdit +=  
954:             new NodeLabelEditEventHandler(OnAfterLabelEdit);  
955:         treeView1.BeforeCheck +=  
956:             new TreeViewCancelEventHandler(OnBeforeCheck);  
957:         treeView1.BeforeCollapse +=  
958:             new TreeViewCancelEventHandler(OnBeforeCollapse);  
959:         treeView1.BeforeExpand +=  
960:             new TreeViewCancelEventHandler(OnBeforeExpand);  
961:         treeView1.BeforeSelect +=  
962:             new TreeViewCancelEventHandler(OnBeforeSelect);  
963:         treeView1.BeforeLabelEdit +=  
964:             new NodeLabelEditEventHandler(OnBeforeLabelEdit);  
965:  
966:  
967:         tvlistBox = new ListBox();  
968:         tvlistBox.Location = new System.Drawing.Point(272, 0);  
969:         tvlistBox.Size = new System.Drawing.Size(304, 424);  
970:         tvlistBox.ForeColor = System.Drawing.SystemColors.WindowText;  
971:         tvlistBox.TabIndex = 1;  
972:  
973:  
974:         button4 = new Button();  
975:         button4.Location = new System.Drawing.Point(16, 376);  
976:         button4.Size = new System.Drawing.Size(96, 24);  
977:         button4.TabIndex = 2;  
978:         button4.Text = "Add Root";  
979:         button4.Click += new EventHandler(OnAddRoot);  
980:  
981:         button5 = new Button();  
982:         button5.Location = new System.Drawing.Point(138, 376);  
983:         button5.Size = new System.Drawing.Size(96, 24);  
984:         button5.TabIndex = 3;  
985:         button5.Text = "Add Child";  
986:         button5.Click += new EventHandler(OnAddChild);  
987:         button5.Enabled=false;  
988:  
989:         TreeTabPage.Controls.Add(button4);  
990:         TreeTabPage.Controls.Add(button5);  
991:         TreeTabPage.Controls.Add(tvlistBox);  
992:         TreeTabPage.Controls.Add(treeView1);  
993:     }  
994:  
995:  
996:
```

LISTING 3.2.10 Continued

NOTE

The application puts it all together by calling all the initializers and adding all the tabs to the main page.

```
997:         public MungoTabApp()
998:     {
999:         //      components = new System.ComponentModel.Container();
1000:        AutoScaleBaseSize = new System.Drawing.Size(5, 13);
1001:        ClientSize = new System.Drawing.Size(600, 477);
1002:
1003:        MainTabControl = new TabControl();
1004:
1005:        MainTabControl.Location = new System.Drawing.Point(8, 8);
1006:        MainTabControl.SelectedIndex = 0;
1007:        MainTabControl.Size = new System.Drawing.Size(584, 448);
1008:        MainTabControl.TabIndex = 0;
1009:
1010:        InitWelcome();
1011:        InitSimple();
1012:        InitLists();
1013:        InitDynamic();
1014:        InitMouse();
1015:        InitTree();
1016:
1017:        mainMenu1 = new MainMenu();
1018:        Menu = mainMenu1;
1019:        Text = "Mungo Tab App";
1020:        timer1 = new Timer();
1021:
1022:        MainTabControl.Controls.Add(SimpleTabPage);
1023:        MainTabControl.Controls.Add(ListBoxTabPage);
1024:        MainTabControl.Controls.Add(DynamicTabPage);
1025:        MainTabControl.Controls.Add(MouseTabPage);
1026:        MainTabControl.Controls.Add(TreeTabPage);
1027:
1028:        Controls.Add(MainTabControl);
1029:
1030:    }
1031:
```

LISTING 3.2.10 Continued**NOTE**

Finally, we add the Main static function that bootstraps the whole process.

```
1032:     public static int Main(string[] args)
1033:     {
1034:         Application.Run(new MungoTabApp());
1035:         return 0;
1036:     }
1037: }
1038: }
```

Summary

This chapter has seen you progress from the simple Windows Forms Hello World application to the tabbed application shown in Listing 3.2.10. We've covered a lot of ground but have really only just begun with Windows Forms. The basic information in this chapter will allow you to place forms and controls, add handlers, and understand the dynamic nature of the new Windows framework.

3.2

USER INTERFACE
COMPONENTS

IN THIS CHAPTER

- Data Binding Strategies 278
- Data Binding Sources 278
- Simple Binding 279
- Simple Binding to a DataSet 283
- Complex Binding of Controls to Data 289
- Binding Controls to Databases Using ADO.NET 293
- Creating a Database Viewer with Visual Studio and ADO.NET 294

In the previous chapter, we looked at some of the more common user interface elements and how to place and interact with them. The user interfaces presented in Windows Forms are not so different from the familiar MFC ones with which we are familiar. However, the MFC controls can require a lot of programming to make them useful. Some of the Windows Forms elements, such as the `TreeView` and `ListView` controls, are very complex and look as if they require a lot of coding to populate and use. This is not always the case, especially when controls are tied to database tables through data binding.

Data Binding Strategies

Data is generally held in tables that have one or more rows and one or more columns. *Data Binding* is the technique of tying user interface controls directly to individual elements, whole rows, whole columns, or entire tables for the purpose of viewing, manipulating, or editing them.

There are two types of data binding. *Simple Binding* ties a simple data viewer such as a text box or a numeric edit box to a single data item in a data table. *Complex Binding* presents a view of a whole table or many tables.

Data Binding Sources

Suitable sources for data binding include any object that exposes the `IList` interface. Most of the .NET collections implement this interface. Data objects provided by ADO.NET also expose `Ilist`, and you can also write your own objects that expose data through this interface.

The `IList` Interface

The `Ilist` interface has only a few methods. They are as follows:

- `Add` adds an item to the list.
- `Clear` removes all items from a list.
- `Contains` searches a list for a certain value.
- `IndexOf` determines the index of a particular value within the list.
- `Insert` places a value into the list before a given index. This moves all following indexes up one and makes the list longer.
- `Remove` takes an item out of the list.
- `RemoveAt` removes one item from a specific index.

Accessing items within the list is done by index. For example, `Item x = myList[n]` items are assumed to be objects so any data can be stored in a list if it's a boxed value type or if it's a .NET object.

Some .NET Objects That Implement `IList`

There are a lot of classes in the framework that implement `IList`. Some of them are just for accessing data. The `Array`, `ArrayList`, and `StringCollection` classes are very commonly used. More complex classes include `DataSetView` and `DataView`, which work in conjunction with ADO to provide data table access.

The less mundane classes that support the interface are used extensively by the .NET developer tools for looking at the metadata associated with classes. These include the `CodeClassCollection` and the `CodeStatementCollection` classes and other objects that use the code Document Object Model or DOM.

Simple Binding

As previously mentioned, simple binding is the business of taking a simple chunk of data and tying it to a user control. When the data changes, the control reflects that change. When the control's display is edited, the data changes if write access is enabled.

Each Windows Forms control maintains a `BindingContext` object that, in turn, has a collection of `CurrencyManager` objects. By the way, the `CurrencyManager` has nothing to do with financial exchanges. It simply maintains a current position within a particular data object for you. Child controls, such as the `GroupBox`, can also have their own independent `BindingContexts` and `CurrencyManagers`.

The illustration in Figure 3.3.1 shows the relationships between Form, Control, `BindingContext` and `CurrencyManager`.

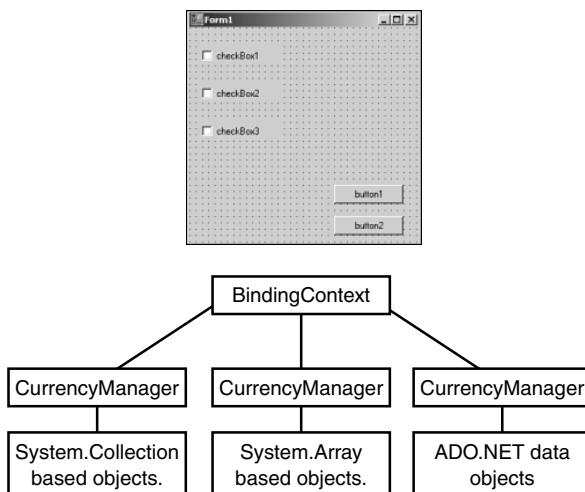


FIGURE 3.3.1

The relationship between form, BindingManager, and CurrencyManager.

Let's take a look at using the `BindingManager` and `CurrencyManager` with a simple application that binds a text box's `Text` property to a collection of strings. Listing 3.3.1 shows the `DataBound` application source code.

LISTING 3.3.1 databound.cs: Simple Binding of Controls to Data

```
1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Collections;
6:     using System.Collections.Specialized;
7:     using System.ComponentModel;
8:     using System.Windows.Forms;
9:
10:    /// <summary>
11:    ///      Summary description for databound.
12:    /// </summary>
13:    public class databound : System.Windows.Forms.Form
14:    {
15:        private Button RightButton;
16:        private Button LeftButton;
17:        private TextBox textBox2;
18:        private TextBox textBox1;
19:        private Button DoneButton;
20:
21:        private StringCollection sa;
22:
23:        public databound()
24:        {
25:            this.textBox2 = new TextBox();
26:            this.RightButton = new Button();
27:            this.textBox1 = new TextBox();
28:            this.DoneButton = new Button();
29:            this.LeftButton = new Button();
30:            this.SuspendLayout();
31:            //
32:            // textBox2
33:            //
34:            this.textBox2.Location = new Point(184, 16);
35:            this.textBox2.Name = "textBox2";
36:            this.textBox2.TabIndex = 2;
37:            this.textBox2.Text = "textBox2";
38:            //
39:            // RightButton
```

LISTING 3.3.1 Continued

```
40:    //
41:    this.RightButton.Location = new Point(192, 64);
42:    this.RightButton.Name = "RightButton";
43:    this.RightButton.TabIndex = 4;
44:    this.RightButton.Text = ">>";
45:    this.RightButton.Click += new EventHandler(this.RightButton_Click);
46:    //
47:    // textBox1
48:    //
49:    this.textBox1.Location = new Point(8, 16);
50:    this.textBox1.Name = "textBox1";
51:    this.textBox1.TabIndex = 1;
52:    this.textBox1.Text = "textBox1";
53:    //
54:    // DoneButton
55:    //
56:    this.DoneButton.Location = new Point(104, 64);
57:    this.DoneButton.Name = "DoneButton";
58:    this.DoneButton.TabIndex = 0;
59:    this.DoneButton.Text = "Done";
60:    this.DoneButton.Click += new EventHandler(this.DoneButton_Click);
61:    //
62:    // LeftButton
63:    //
64:    this.LeftButton.Location = new Point(16, 64);
65:    this.LeftButton.Name = "LeftButton";
66:    this.LeftButton.TabIndex = 3;
67:    this.LeftButton.Text = "<<";
68:    this.LeftButton.Click += new EventHandler(this.LeftButton_Click);
69:    //
70:    // databound
71:    //
72:    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
73:    this.ClientSize = new System.Drawing.Size(294, 111);
74:    this.ControlBox = false;
75:    this.Controls.AddRange(new Control[] {
76:        this.RightButton,
77:        this.LeftButton,
78:        this.textBox2,
79:        this.textBox1,
80:        this.DoneButton});
81:    this.FormBorderStyle = FormBorderStyle.FixedDialog;
82:    this.Name = "databound";
83:    this.ShowInTaskbar = false;
```

LISTING 3.3.1 Continued

```
84:         this.Text = "databound";
85:         this.ResumeLayout(false);
86:
87:         //This is the setup code for the simple binding example
88:         //We have bound two text controls to the same StringCollection
89:
90:         // This is setting up the "database" (an IList thing)
91:         sa=new StringCollection();
92:         sa.Add("Hello databinding");
93:         sa.Add("Sams publishing");
94:         sa.Add("C# example");
95:         sa.Add("By Bob Powell");
96:
97:         //This binds the controls to that database.
98:         this.textBox1.DataBindings.Add("Text",sa,"");
99:         this.textBox2.DataBindings.Add("Text",sa,"");
100:        //See the button handlers below for details
101:        //on how to move through the data.
102:    }
103:
104:    /// <summary>
105:    ///     Clean up any resources being used.
106:    /// </summary>
107:    public override void Dispose()
108:    {
109:        base.Dispose();
110:    }
111:
112:
113:    //Very simply increments the Position property of the CurrencyManager
114:    protected void RightButton_Click (object sender, System.EventArgs e)
115:    {
116:        //Note how updating one position affects all
117:        //controls bound to this data
118:        this.BindingContext[sa].Position++;
119:    }
120:
121:    //Very simply decrements the Position property of the CurrencyManager
122:    protected void LeftButton_Click (object sender, System.EventArgs e)
123:    {
124:        //Note how updating one position affects all
125:        //controls bound to this data
126:        this.BindingContext[sa].Position--;
127:    }
```

LISTING 3.3.1 Continued

```
128:  
129:    protected void DoneButton_Click (object sender, System.EventArgs e)  
130:    {  
131:        Application.Exit();  
132:    }  
133:  
134:    public static void Main()  
135:    {  
136:        Application.Run(new databound());  
137:    }  
138:    }  
139: }
```

Compile this program with the command line:

```
csc t:/winexe databound.cs
```

The setup of the objects on the form will be familiar to you so we don't need to re-iterate those principles. The interesting parts are on lines 91–95 which creates and populates a simple `StringCollection`. Remember that the .NET collections all implement the `IList` interface, making them candidates for databinding. Lines 98 and 99 perform the actual binding of the data to the textboxes. Finally, the click handlers on lines 114–119 and 122–127 move forward or backward through the data by moving the position, analogous to the cursor in a database, through the data.

3.3

DATA BOUND
CONTROLS

Simple Binding to a DataSet

Binding of all sorts of controls is an important feature of .NET. Databound controls are available in Windows Forms, WebForms, and ASP.NET. All these controls can use ADO.NET to bind to database tables either on the local machine or somewhere else on the network. The vehicle for database binding is the `DataSet`. This class represents a set of cached data that is held in memory. The data cache can be created manually or populated with data drawn from a database using ADO.NET's database connection system. After the information is in that memory store on your machine, you can bind controls to individual columns of data and select rows for editing or viewing.

The easiest way to understand a `DataSet` is to use it. Listing 3.3.2 shows an application that generates and manipulates a simple `DataSet` on your machine using simple binding to controls.

LISTING 3.3.2 datasetapp.cs: Simple Binding of Multiple Controls

```
1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Collections;
6:     using System.Data;
7:     using System.ComponentModel;
8:     using System.Windows.Forms;
9:
10:    //This application shows simple data binding to a
11:    //programmatically created dataset.
12:    public class datasetapp : System.Windows.Forms.Form
13:    {
14:        private System.ComponentModel.Container components;
15:
16:        //Component declarations
17:        private Label lbl_first, lbl_name, lbl_title, lbl_company, lbl_phone;
18:        private TextBox FirstName, SurName, Title, Company, Phone;
19:        private Button btnNext, btnPrev, btnNew, btnEnd;
20:
21:        //The dataset used to store the table
22:        private DataSet dataset;
23:
24:        //Button handler to navigate backwards through the table records
25:        private void OnPrev(Object sender, EventArgs e)
26:        {
27:            this.BindingContext[dataset.Tables["Contacts"]].Position--;
28:        }
29:
30:        //Button handler to navigate forward through the table records
31:        private void OnNext(Object sender, EventArgs e)
32:        {
33:            this.BindingContext[dataset.Tables["Contacts"]].Position++;
34:        }
35:
36:        //Button handler to create a new row
37:        private void OnNew(Object sender, EventArgs e)
38:        {
39:            NewEntry();
40:        }
41:
42:        //Button handler to exit the application
43:        private void OnEnd(Object sender, EventArgs e)
44:        {
```

LISTING 3.3.2 Continued

```
45:     Application.Exit();
46: }
47:
48: //Method to move to the last record. Used when adding a row.
49: private void MoveToEnd()
50: {
51:     this.BindingContext[dataset.Tables["Contacts"]].Position=
52:         dataset.Tables["Contacts"].Rows.Count-1;
53: }
54:
55: //Method to add a new row to the table. Called at initialization
56: //and by the "New" button handler.
57: private void NewEntry()
58: {
59:     DataRow row = dataset.Tables["Contacts"].NewRow();
60:     //set up row data with new entries of your choice
61:     row["First"]="Blank";
62:     row["Name"]="";
63:     row["Company"]="";
64:     row["Title"]="";
65:     row["Phone"]="";
66:     dataset.Tables[0].Rows.Add(row);
67:     dataset.AcceptChanges();
68:     MoveToEnd();
69: }
70:
71: //Called at creation to initialize the
72: //dataset and create an empty table
73: private void InitDataSet()
74: {
75:     dataset = new DataSet("ContactData");
76:
77:     DataTable t=new DataTable("Contacts");
78:
79:     t.Columns.Add("First",typeof(System.String));
80:     t.Columns.Add("Name",typeof(System.String));
81:     t.Columns.Add("Company",typeof(System.String));
82:     t.Columns.Add("Title",typeof(System.String));
83:     t.Columns.Add("Phone",typeof(System.String));
84:
85:     t.MinimumCapacity=100;
86:
87:     dataset.Tables.Add(t);
88: }
```

3.3**DATA BOUND
CONTROLS**

LISTING 3.3.2 Continued

```
89:
90:        //Called at initialization to do simple binding of the edit
91:        //controls on the form to the dataset's "Contacts" table entries
92:        private void BindControls()
93:        {
94:            FirstName.DataBindings.Add("Text",dataset.Tables["Contacts"],
95:                                     "First");
95:            Surname.DataBindings.Add("Text",dataset.Tables["Contacts"],
96:                                     "Name");
96:            Title.DataBindings.Add("Text",dataset.Tables["Contacts"],
97:                                     "Title");
97:            Company.DataBindings.Add("Text",dataset.Tables["Contacts"],
98:                                     "Company");
98:            Phone.DataBindings.Add("Text",dataset.Tables["Contacts"],
99:                                     "Phone");
99:        }
100:
101:       //Constructor. Positions the form controls,
102:       //Initializes the dataset, binds the controls and
103:       //wires up the handlers.
104:       public datasetapp()
105:       {
106:           this.components = new System.ComponentModel.Container ();
107:           this.Text = "datasetapp";
108:           this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
109:           this.ClientSize = new System.Drawing.Size (250, 200);
110:           this.FormBorderStyle = FormBorderStyle.Fixed3D;
111:
112:           lbl_first = new Label();
113:           lbl_first.Text="First name";
114:           lbl_first.Location = new Point(5,5);
115:           lbl_first.Size = new Size(120,28);
116:           lbl_first.Anchor = AnchorStyles.Left | AnchorStyles.Right;
117:           Controls.Add(lbl_first);
118:
119:           FirstName = new TextBox();
120:           FirstName.Location = new Point(125,5);
121:           FirstName.Size = new Size(120,28);
122:           FirstName.Anchor = AnchorStyles.Left | AnchorStyles.Right;
123:           Controls.Add(FirstName);
124:
125:           lbl_name = new Label();
126:           lbl_name.Text="Surname";
127:           lbl_name.Location = new Point(5,35);
```

LISTING 3.3.2 Continued

```
128:         lbl_name.Size = new Size(120,28);
129:         lbl_name.Anchor = AnchorStyles.Left|AnchorStyles.Right;
130:         Controls.Add(lbl_name);
131:
132:         SurName = new TextBox();
133:         SurName.Location = new Point(125,35);
134:         SurName.Size = new Size(120,28);
135:         SurName.Anchor = AnchorStyles.Left | AnchorStyles.Right;
136:         Controls.Add(SurName);
137:
138:         lbl_company = new Label();
139:         lbl_company.Text="Company";
140:         lbl_company.Location = new Point(5,65);
141:         lbl_company.Size = new Size(120,28);
142:         Controls.Add(lbl_company);
143:
144:         Company = new TextBox();
145:         Company.Location = new Point(125,65);
146:         Company.Size = new Size(120,28);
147:         Controls.Add(Company);
148:
149:         lbl_title = new Label();
150:         lbl_title.Text="Title";
151:         lbl_title.Location = new Point(5,95);
152:         lbl_title.Size = new Size(120,28);
153:         Controls.Add(lbl_title);
154:
155:         Title = new TextBox();
156:         Title.Location = new Point(125,95);
157:         Title.Size = new Size(120,28);
158:         Controls.Add(Title);
159:
160:         lbl_phone = new Label();
161:         lbl_phone.Text="Telephone";
162:         lbl_phone.Location = new Point(5,125);
163:         lbl_phone.Size = new Size(120,28);
164:         Controls.Add(lbl_phone);
165:
166:         Phone = new TextBox();
167:         Phone.Location = new Point(125,125);
168:         Phone.Size = new Size(120,28);
169:         Controls.Add(Phone);
170:
171:         btnNew = new Button();
```

LISTING 3.3.2 Continued

```
172:             btnNew.Location = new Point(5,155);
173:             btnNew.Size = new Size(70,28);
174:             btnNew.Text="New";
175:             btnNew.Click+=new EventHandler(OnNew);
176:             Controls.Add(btnNew);
177:
178:             btnPrev = new Button();
179:             btnPrev.Location = new Point(80,155);
180:             btnPrev.Size = new Size(35,28);
181:             btnPrev.Text="<<";
182:             btnPrev.Click += new EventHandler(OnPrev);
183:             Controls.Add(btnPrev);
184:
185:             btnEnd = new Button();
186:             btnEnd.Location = new Point(120,155);
187:             btnEnd.Size = new Size(70,28);
188:             btnEnd.Text="End";
189:             btnEnd.Click += new EventHandler(OnEnd);
190:             Controls.Add(btnEnd);
191:
192:             btnNext = new Button();
193:             btnNext.Location = new Point(200,155);
194:             btnNext.Size = new Size(35,28);
195:             btnNext.Text=">>";
196:             btnNext.Click += new EventHandler(OnNext);
197:             Controls.Add(btnNext);
198:
199:             InitDataSet();
200:
201:             NewEntry();
202:
203:             BindControls();
204:
205:         }
206:
207:         //Cleans up the Form
208:         public override void Dispose()
209:         {
210:             base.Dispose();
211:             components.Dispose();
212:         }
213:
214:         //Main method to instantiate and run the application.
215:         static void Main()
```

LISTING 3.3.2 Continued

```

216:         {
217:             Application.Run(new datasetapp());
218:         }
219:     }
220: }
```

Let's examine Listing 3.3.2 in more detail. The main initialization of the DataSet takes place on lines 73–88. Here, a DataSet is created and a new table placed in its tables collection. The table is given a set of columns that are identified by the strings "First", "Name", and so on.

Adding data to a table is done one row at a time. The method on lines 57–69 adds a new row to the end of the table and optionally fills it with text. You could also create a table with numeric values or other types.

Lines 92–99 do the binding of the data in the table to the `Text` property of the identified controls. These controls can now be used to edit the information in the table, and they will display each entry automatically as you navigate through the database.

The button handlers that do the navigation are on lines 24–37. These use the `ContextManager.Position` property to index the rows in the table.

Figure 3.3.2 shows a screenshot of the application.

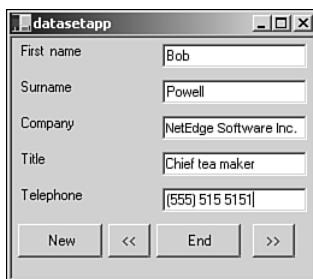
3.3**DATA BOUND
CONTROLS**

FIGURE 3.3.2

The DataSet simple binding application.

Complex Binding of Controls to Data

Simple binding is the process of tying a single control to a single data item. Complex binding ties a single control to many data items. Controls that are often used for complex data binding include `DataGrid` which has the appearance of a simple spreadsheet. This control can display, edit, or add and delete records from a table. A single databinding operation is used to attach the

control to a DataTable. The DataGridView can only be used to show one table at a time, but it can be used to show relationships between tables.

Listing 3.3.3 shows a modified version of the simple binding example demonstrated in Listing 3.3.2. This listing removes all the controls associated with the DataSet and replaces them with a complex bound grid control. Don't be put off by the title. Complex data binding is complex only in its function, not its usage.

LISTING 3.3.3 gridbind.cs: An Example of Complex Data Binding

```
1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Collections;
6:     using System.Data;
7:     using System.ComponentModel;
8:     using System.Windows.Forms;
9:
10:    //This application shows simple data binding to a
11:    //programmatically created dataset.
12:    public class GridBind : System.Windows.Forms.Form
13:    {
14:        private System.ComponentModel.Container components;
15:        private System.Windows.Forms.DataGrid dataGrid1;
16:
17:        //The dataset used to store the table
18:        private DataSet dataset;
19:
20:        //Called at creation to initialize the
21:        //dataset and create an empty table
22:        private void InitDataSet()
23:        {
24:            dataset = new DataSet("ContactData");
25:
26:            DataTable t=new DataTable("Contacts");
27:
28:            t.Columns.Add("First",typeof(System.String));
29:            t.Columns.Add("Name",typeof(System.String));
30:            t.Columns.Add("Company",typeof(System.String));
31:            t.Columns.Add("Title",typeof(System.String));
32:            t.Columns.Add("Phone",typeof(System.String));
33:
34:            t.MinimumCapacity=100;
35:
```

LISTING 3.3.3 Continued

```
36:     dataset.Tables.Add(t);
37: }
38:
39: //Called at initialization to do complex binding of the DataGridView
40: //to the dataset's "Contacts" table entries
41: private void BindGrid()
42: {
43:     this.dataGridView1.SetDataBinding(dataset.Tables["Contacts"], "");
44: }
45:
46: //Constructor. Positions the form controls,
47: //Initializes the dataset, binds the controls and
48: //wires up the handlers.
49: public GridBind()
50: {
51:     InitializeComponent();
52:
53:     InitDataSet();
54:
55:     BindGrid();
56:
57: }
58:
59: //Cleans up the Form
60: public override void Dispose()
61: {
62:     base.Dispose();
63:     components.Dispose();
64: }
65:
66: //Method added by the form designer
67: private void InitializeComponent()
68: {
69:     this.components = new System.ComponentModel.Container ();
70:     this.dataGridView1 = new System.Windows.Forms.DataGridView ();
71:     dataGridView1.BeginInit ();
72:     dataGridView1.Location = new System.Drawing.Point (8, 16);
73:     dataGridView1.Size = new System.Drawing.Size (472, 224);
74:     dataGridView1.DataMember = "";
75:     dataGridView1.TabIndex = 0;
76:     this.Text = "GridBind";
77:     this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
78:     this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.Fixed3D;
79:     this.ClientSize = new System.Drawing.Size (486, 251);
```

3.3**DATA BOUND
CONTROLS**

LISTING 3.3.3 Continued

```
80:         this.Controls.Add (this.dataGrid1);
81:         dataGridView1.EndInit ();
82:     }
83:
84:
85:
86:     //Main method to instantiate and run the application.
87:     static void Main()
88:     {
89:         Application.Run(new GridBind());
90:     }
91: }
92: }
```

Compile this program with the following command line:

```
csc /t:winexe gridbind.cs
```

Looking at the information in Listing 3.3.3, you can see that the code used to navigate the data set is no longer needed. The method to add a new row of data is not needed either. The **DataGrid** control does all that for you. Binding the data to the grid takes place on line 43. Initialization of the data set is identical to the previous example; it is shown on lines 22–37. Figure 3.3.3 shows the **GridBind** application in action. Notice that the grid provides column and row headers that allow you to select a whole row or order the table according to column alphabetic order.

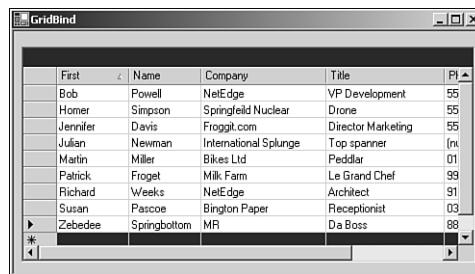


FIGURE 3.3.3

Complex binding to a DataGrid.

The **DataGrid** control can also be used to show hierarchical relationships between multiple data tables. Perhaps a Customer Resource Management application would want to show the relationship between a customer and his or her orders or a client and his or her technical support incidents. Complex binding of data makes this otherwise difficult process painless.

So far, you have seen data binding to `DataSets` and `DataTables` created on-the-fly by the listing examples. A good real world example will be to show complex binding to an existing database.

The objects we have been using to store and manipulate the data are parts of the ADO.NET framework. This system allows high level interaction with database tables from many different database providers. ADO.NET can be used in conjunction with OleDb, SQL, and other database drivers. Take a look now at an example of complex bound database usage.

Binding Controls to Databases Using ADO.NET

ADO.NET employs a disconnected database model that uses data tables on the client machine to cache information while it is viewed or manipulated. Even when the database is stored on the same machine as the editing program, ADO.NET will copy a snapshot of data into a `DataSet` and pass changes back to the database later.

This model is used to allow better scaling of the database application over a network because it obviates the need for many, simultaneous, open connections to the database.

Data tables are very structured and hierarchical. This makes XML an ideal medium for transmitting and storing data. Indeed, when ADO makes a connection to a database, the information will be transported as XML. This means that your software can take advantage of intranet and Internet data stores just as easily as it can local databases.

It is important to note that transporting large amounts of data to and from databases becomes more demanding as the database becomes more disconnected from the client. Modern n-tier systems often employ many databases in different geographical locations and Web services to provide data on demand to many clients. This means that your client application should try to use small chunks of data that can be refreshed quickly and with as little strain as possible on network bandwidth. For this reason, many modern data systems employ a messaging architecture that deals with data transactions at a very granular level. It often has been the responsibility of the application programmer to devise and manage schemes for such transactions, but the high level abilities of ADO.NET hide much of that pain from you and allow you to be more productive with data.

Visual Studio Data is copied to or from `DataSet` by a `DataAdapter`. ADO.NET provides two main `DataAdapter` types, the `OleDbDataAdapter` that connects `DataSets` to OLE DB data providers and the `SqlDataAdapter`, targeting SQL server connections. Both of these data adapter classes use a set of `Command` objects to perform selects, updates, inserts, and deletes in the databases to which they connect. The `Select` command gets data from the data source. The `Update`, `Delete`, and `Insert` commands all change the data source based on changes made to the `DataSet`.

Filling a DataSet with information from a database is fairly simple. The steps to perform are shown in the code snippet that follows:

```
1: System.Data.OleDb.OleDbConnection connection =
2:   new System.Data.OleDb.OleDbConnection();
3: connection.ConnectionString=
4: @"Provider=Microsoft.Jet.OLEDB.4.0;" +
5: @"Data Source=northwind.mdb; ";
6: System.Data.OleDb.OleDbCommand SelectCommand=
7:   new System.Data.OleDb.OleDbCommand("SELECT * FROM Customers");
8: SelectCommand.Connection=connection;
9: System.Data.DataSet ds=new System.Data.DataSet("Customer Table");
10: ds.Tables.Add("Customers");
11: System.Data.OleDb.OleDbDataAdapter adapter =
12:   new System.Data.OleDb.OleDbDataAdapter(SelectCommand);
13: adapter.Fill(ds);
14: Console.WriteLine(ds.GetXml());
```

This snippet assumes that the NortWind database `northwind.mdb` is available in the local directory.

Lines 1–5 create an OleDbConnection and set the connection string.

Lines 6 and 7 create the SelectCommand and populate the select string with a valid query.

Line 8 associates the database connection with the select command, while lines 9 and 10 create a DataSet object and add one table to it.

Lines 11 and 12 create a data adapter and add the select command. Line 13 fills the data set from the adapter and finally, to prove it actually did something, the last line dumps the whole northwing customer table from the DataSet as XML.

From the point of view of your own productivity, you will most likely perform such an operation using the Visual Studio.NET IDE to create the application. Databinding using Visual Studio generates some enormous blocks of code and does a very comprehensive job of connecting up your database. We have generally avoided using Visual Studio until now because the layout of applications created in this manner is not easily read and there are sometimes dire consequences when you edit the code generated by the studio. In this case, however, a good exercise will be to do a complete walk through of a database viewer using the IDE and then examine the resulting application to see what it does.

Creating a Database Viewer with Visual Studio and ADO.NET

Follow these steps carefully to create the application.

1. Create a new Windows Forms project, and call it **DBBind**.
2. Drag a DataGrid from the toolbox and position it on the form. Select the DataGrid properties and set the Dock property to Fill by clicking the center button in the UI editor that pops up for the Dock property.
3. From the toolbar, select an **OleDbDataAdapter** from the Data tab. When you place it on the form, you'll see the Data Adapter Configuration Wizard dialog.
4. On the first page of the wizard, select the Northwind database. You may need to click New Connection, select Microsoft Jet 4.0 OLE DB Provider in the Provider tab, press Next, and then browse for **northwind.mdb**. When you find it, or another database with which you are familiar, press the Test Connection button to ensure that the database is useable. Press OK on the New Connection dialog and press Next on the wizard.
5. Select the radio button marked Use SQL Statements. Press Next.
6. Type in a query that you know or select the query builder to create queries from the tables in the connected database. For our example, we used the query builder, added the **Customers** table from **NorthWind**, selected the check box marked All Columns on the table shown, and pressed OK on the query builder dialog.
7. Pressing Next shows you the successful creation of the adapter and the connection, and you can click the Finish button. At this point, you will have a form with two objects, an **OleDbDataAdapter** and an **OleDbConnection** in the icon tray.
8. Select the **OleDbDataAdapter** in the icon tray. Below the property browser, you will see a link that reads Generate DataSet. Click this link to create a dataset. Simply click OK on the following dialog; the default values are good. Now you'll have three icons in the tray.
9. Select the DataGrid on the form again and edit the **DataSource** property using the drop-down combo box provided. You will see several data sources. If you have followed along and used the **NorthWind** database, select the **dataSet11.Customers** choice. When you do this, you'll see the DataGrid on the form begin to display all the column headers for the database.
10. The final step is a manual one. You need to edit the constructor of the class to add the **Fill** command. Add the following line of code after the **InitializeComponent** call in the class constructor;

```
this.oleDbDataAdapter1.Fill(this.dataSet11);
```

Now you can build and run the program by pressing F5.

You will see the application shown in Figure 3.3.4, which will allow you to browse, order, edit, and generally muck-about with the **NorthWind** **Customers** table.

3.3

DATA BOUND CONTROLS

	Address	City	CompanyName	ContactName	ContactTitle	Country	CustomerID	Fax	Phone	PostalCode
▶	Ober str. 57	Berlin	Alfreds Futterkiste	Maria Anders	Sales Repesentative	Germany	ALFKI	030-0076545	030-0074321	12209
	Avda. de la Constitución 9, 23456	México D.F.	Ana Trujillo Empresarial	Ana Trujillo	Owner	Mexico	ANATR	(5) 555-3745	(5) 555-4729	05021
	Madamadores 2	México D.F.	Antonio Moreno	Antonio Moreno	Owner	Mexico	ANTON	(null)	(5) 555-3932	05023
	120 Hanover	London	Around the Horn	Thomas Harder	Sales Repesentative	UK	AROUT	(171) 555-67	(171) 555-77	WA1
	Berguvsvägen 22	Luleå	Berglunds snabbköp	Christina Berglund	Order Adminstrator	Sweden	BERGS	0921-12 34 6	0921-12 34 6	S-901 70
	Fonterstr. 57	Mannheim	Blauer See Delicatessen	Hanna Moen	Sales Repesentative	Germany	BLAUS	0621-08924	0621-08460	66300
	24, place Kléber	Strasbourg	Blondel pâtisserie	Frédérique Cuvée	Marketing Manager	France	BLOND	88.60.15.32	88.60.15.31	67000
	C/C Aragó, 67	Madrid	Bólido Comidas Preparadas	Martín sommerville	Owner	Spain	BOLID	(91) 555 91 9	(91) 555 22 8	28023
	12, rue des Bouchers	Marseille	Bon app'	Laurence Lebœuf	Owner	France	BONAP	91.24.45.41	91.24.45.40	13008
	23 Tsalwassee	Tsalwassee	Bottom-Dollar Clothing	Elizabeth Lin	Accounting Manager	Canada	BOTTM	(604) 555-37	(604) 555-47	T2F 8L
	Fauntleroy Ci	London	B's Beverages	Victoria Ashworth	Sales Repesentative	UK	BSBEV	(null)	(171) 555-12	EC2 5
	Cerro 333	Buenos Aires	Cactus Comercio Exterior	Patricia Simpson	Sales Agent	Argentina	CACTU	(1) 135-4892	(1) 135-5555	1010
	Sierras de Grado	México D.F.	Centro comercial Moll	Francisco Chávez	Marketing Manager	Mexico	CENTC	(5) 555-7293	(5) 555-3395	05022
	Hauptstr. 29	Bern	Chop-suey Chinese	Yang Wang	Owner	Switzerland	CHOPS	(null)	0452-076545	3012
	Av. dos Lusíadas	São Paulo	Comércio Mineiro	Pedro Afonso	Sales Associate	Brazil	COMMI	(null)	(11) 555-764	05432
	Berkeley Gar	London	Consolidated Exporters	Elizabeth Brody	Sales Repesentative	UK	CONSH	(171) 555-91	(171) 555-22	WA1
	Walsenweg 2	Aachen	Drachenblut Winery	Sven Ottiebel	Order Adminstrator	Germany	DRACD	0241-059428	0241-039123	52066
	67, rue des Clémants	Nantes	Du monde entier	Jeanine Labrune	Owner	France	DUMON	40.67.89.89	40.67.88.88	44000
	35 King George	London	Eastern Connection	Ann Devonshire	Sales Agent	UK	EASTC	(171) 555-33	(171) 555-02	WC2E 8
	Kirchgasse 6	Graz	Ernst Handel	Roland Mendel	Marketing Manager	Austria	ERNSH	7675-3426	7675-3425	8010
	Rua Orós, 92	São Paulo	Família Arquimarketing	Aria Cruz	Marketing Associate	Brazil	FAMILA	(null)	(11) 555-985	05442
	C/C Moralzarz	Madrid	FISSA Fabrica de Seda	Diego Roel	Accounting Manager	Spain	FISSA	(91) 555 55 9	(91) 555 94 4	28034
	184, chaussée de Lille	Nantes	Folies gourmandes	Marine Rancé	Assistant Sales Representative	France	FOLIG	20.16.10.17	20.16.10.16	58000
	Åkergratan 24	Bräcke	Folk och färg	Maria Larsson	Owner	Sweden	FOLKO	(null)	0695-34 67 2	S-844
	Berliner Platz	München	Frankenversand	Peter Franke	Marketing Manager	Germany	FRANK	089-0877451	089-0877310	80805
	54, rue Royal	Nantes	Francesca restau	Carine Schmid	Marketing Manager	France	FRANR	40.32.21.20	40.32.21.21	44000

FIGURE 3.3.4*Complex DataBinding of the DataGrid to a database.*

As an education, go back into the Visual Studio editor again and find the `InitializeComponent` method. It will probably be folded away in a little region and you'll have to expand it. Take a look at the code added by the DataAdapter wizard to set up the `OleDbDataAdapter` connection and the four database command objects.

As performing the walk through will generate the code for you, we won't list it here and save a small forest or two. You can clearly see that the wizards and helpers in ADO.NET really save your fingers.

Summary

In this chapter, we looked at the basic principals of data binding in .NET and saw how the power of the .NET controls can work for us to create a very productive environment. We saw how simple binding works for single data items and single controls. We then saw how complex binding works for the DataGrid both in isolation with a locally created data table and using a full database through ADO.NET.

We're keeping the focus on Windows Forms now as we continue with Chapter 3.4, "Windows Forms Example Application (Scribble. NET)," which deals with more component usage, including the use of resources as a means of decorating your GUI elements.

Windows Forms Example Application (Scribble .NET)

CHAPTER

3.4

IN THIS CHAPTER

- Resources in .NET 298
- Localization Nuts and Bolts 298
- .NET Resource Management Classes 299
- Creating Text Resources 301
- Using Visual Studio.NET for Internationalization 308
- Image Resources 310
- Using Image Lists 311
- Programmatic Access to Resources 315
- Reading and Writing RESX XML Files 321

In this chapter, we will be looking at some more of the features of Windows Forms that you will need to be familiar with when writing applications for the .NET framework.

Specifically, we'll deal with resources, including text, bitmap images, and icons. We'll also be looking at globalization, the process of making your applications friendly to an international audience.

Resources in .NET

Unlike Win32 and MFC, .NET applications do not rely heavily on resources for dialog and form layout. However, resources are required for localization of applications, integration of images and icons, and for custom data that can be easily modified without needing to recompile an application.

The resource model under .NET is that of a default resource for an application and an optional set of additional resources that provide localization, or more properly internationalization, data for a specific culture. For example, an application might have a default resource culture of US—English but require localization for the UK—English, where the currency symbol would be different, or for French, where the text strings associated with menus and dialogs would be in the correct language.

The default resource is usually contained within the assembly that holds the application code. Other assemblies, called *satellite assemblies*, have resources that can be loaded and used when needed. This approach is similar to the idea of an application using a resource DLL under Win32 or MFC.

A good “global citizen” application should always be localizable and should always provide a resource for any culture in which that the software is used. It is also true that not every application needs to be a global citizen, so you must choose your level of culture—friendliness—according to your distribution and target audience needs.

Localization Nuts and Bolts

It's good practice to build your applications with localization in mind. The most important thing you can do to prepare for this is to not embed any strings in your code that the user will see. For example you would never use:

```
MessageBox.Show("Hello World!");
```

because it embeds the “Hello World!” string in the source code. To make this program a good global citizen, you would create a default resource where the string was identified by a value. For example,

```
MessageBox.Show((String)resources.GetObject("MyHelloWorldResourceString"));
```

In this line, the actual text of the message is not placed in the code but retrieved from a resource that has a string identified as "MyHelloWorldResourceString".

If your default culture is US-English, the string would say “Hello World!” If the application needed to be used in France, a satellite assembly with the French resources would have a string with the same "MyHelloWorldResourceString" ID, except that this string would have the value of “Bonjour Le Monde!”

.NET Resource Management Classes

The .NET framework provides a family of classes that are designed to make resource management consistent and easy. These classes are as follows:

- `ResourceManager` is used to manage all resources in an application.
- `ResourceSet` represents a collection of all the resources for a particular culture.
- `ResourceReader` is an implementation class for the `IResourceReader` interface. This is used to read resources from a stream.
- `ResourceWriter` is an implementation of the `IResourceWriter` interface. This are used to write resources to a stream.

Your application must use a `ResourceManager` to load the resource assemblies and make them available to the rest of the program. The `ResourceManager` will return a `ResourceSet` for a given culture.

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

Got Culture?

Different cultures and languages are identified through the `CultureInfo` class. This class is a simple information store that contains data about the culture selected by the user when he or she installed the operating system. It contains information about the language, calendar, currency, and other preferences that the user might have chosen. Your software will have access to this information so that it can determine whether it has the ability to offer culture-specific resources or must choose the neutral default value.

Being culture specific might be very important in the message you are trying to deliver. There is an anecdote that illustrates why.

A Little Bit of Culture

A large soap manufacturer created a printed advertisement for display on billboards that showed dirty clothes being loaded into a washing machine. A second picture showed the washing powder being poured into the machine and a third image depicted these clothes being taken out of the washer all clean and fresh. They couldn't understand why the sales of the product dropped dramatically in certain countries until it was pointed out that the places that had low sales were right-to-left reading cultures. Their advertisement, created by American's, had shown clean clothes washed in this soap powder emerging all dirty and stained from the machine. Never underestimate the power of assumption.

Culture information about languages is provided by a two-letter code and an optional region identifier that accompanies the two letter code. For example, the neutral English language is represented by the two letter code en, but the different flavors of English—English from England, Canada, the United States and other places—is identified by the sub-codes GB, CA, US, and so on.

Listing 3.4.1 shows a program that will get the `CultureInfo` for your computer and display the language and region that applies to your computer. It also shows the full name of the culture and which calendar your current culture is using.

LISTING 3.4.1 culture.cs: Displaying the Current Culture Information

```
1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Collections;
6:     using System.ComponentModel;
7:     using System.Globalization;
8:     using System.Windows.Forms;
9:     using System.Data;
10:
11: class CultureInfoTest : Form
12: {
13:
14:     public CultureInfoTest()
15:     {
```

LISTING 3.4.1 Continued

```
16:     this.Size = new Size(300,200);
17:
18:     Label l = new Label();
19:     l.Location = new Point(3,5);
20:     l.Size = new Size(294,190);
21:
22:     InputLanguage inL = InputLanguage.CurrentInputLanguage;
23:
24:     CultureInfo info = inL.Culture;
25:
26:     l.Text = String.Format("Culture identifier = {0}\n"+
27:                           "Display name = {1}\n"+
28:                           "Calender = {2}\n",
29:                           info.ToString(),
30:                           info.DisplayName,
31:                           info.Calendar.ToString());
32:     this.Controls.Add(l);
33:
34: }
35: static void Main()
36: {
37:     Application.Run(new CultureInfoTest());
38: }
39: }
40:
41: }
```

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

Creating Text Resources

There are several ways you can create text resources for your programs. The simplest is by creating a text file that has a set of name-value pairs. These specify a name that you can use in your code to find the resource and the text to associate with that name.

When you've finished making your basic string resource file, you must turn it into a .resx or .resource file using the .NET tool RESGEN.EXE. The following example shows this process.

The String Resource Text

The following simple text file shows a set of name/-value pairs for a resource that will be used to provide the default strings for an application. Note that quotation marks are not required on strings. If you use quotes, they will be embedded in the resource string.

```
#Default culture resources  
WindowText = Internationalization example  
LabelText = Hello World!!!
```

The text file can be converted into one of two forms—an XML-based form that is stored as a .resx file or directly to its compiled .resource form. The .resx file is an XML dataset that is used to serialize resource information during development. Conversion of resources from one form to another is accomplished using the Resgen utility. Type in the previous simple resource text and save it as **firstresource.txt**, and then create a .resx file using the following command line:

```
resgen firstresource.txt firstresource.resx
```

The utility will compile the file and tell you that two resources have been converted. Listing 3.4.2 shows the resulting XML file.

LISTING 3.4.2 firstresource.resx: The Converted Resource File in XML Form

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" targetNamespace="" xmlns="">
    ↵ xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    ↵ xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
      <xsd:element name="root" msdata:IsDataSet="true">
        <xsd:complexType>
          <xsd:choice maxOccurs="unbounded">
            <xsd:element name="data">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="value" type="xsd:string" minOccurs="0"
    ↵ msdata:Ordinal="1" />
                  <xsd:element name="comment" type="xsd:string" minOccurs="0"
    ↵ msdata:Ordinal="2" />
                  </xsd:sequence>
                  <xsd:attribute name="name" type="xsd:string" />
                  <xsd:attribute name="type" type="xsd:string" />
                  <xsd:attribute name="mimetype" type="xsd:string" />
                </xsd:complexType>
              </xsd:element>
            <xsd:element name="resheader">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="value" type="xsd:string" minOccurs="0"
    ↵ msdata:Ordinal="1" />
```

LISTING 3.4.2 Continued

```
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string" use="required" />
</xsd:complexType>
</xsd:element>
</xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:schema>
<data name="WindowText">
    <value>Internationalization example</value>
</data>
<data name="LabelText">
    <value>Hello World!!!</value>
</data>
<resheader name="ResMimeType">
    <value>text/microsoft-resx</value>
</resheader>
<resheader name="Version">
    <value>1.0.0.0</value>
</resheader>
<resheader name="Reader">
    <value>System.Resources.ResXResourceReader</value>
</resheader>
<resheader name="Writer">
    <value>System.Resources.ResXResourceWriter</value>
</resheader>
</root>
```

You can see from Listing 3.4.2 that this is probably not a file you will want to type in by hand. The first part of the file is the schema that defines XML content of the file. Each element is contained within an XML node tagged `<data name="some-name">`. If you look through the file, you will see the two resources that were specified in the text file and their respective values. It is this .resx form of the resources that Visual Studio maintains when you create resources in your applications.

To use the resource file in a sample program, you will need to convert it to its compiled form. You can do this from either the text format or the XML format by using one of the following command lines:

```
Resgen firstresource.txt firstresource.resources
```

or

```
Resgen firstresource.resx firstresource.resources
```

A Simple Winforms Application That Relies on a Resource

To test the resource file shown in Listing 3.4.2, you can create a simple “Hello World”-type application as shown in Listing 3.4.3. This program relies on the `firstresource.resources` file created previously to provide the text for the label and the window title bar.

LISTING 3.4.3 `heliores.cs`: Using Simple String Resources

```
1: namespace Sams
2: {
3:     using System;
4:     using System.Drawing;
5:     using System.Collections;
6:     using System.ComponentModel;
7:     using System.Globalization;
8:     using System.Windows.Forms;
9:     using System.Data;
10:    using System.Resources;
11:
12:    class heliores: Form
13:    {
14:
15:        public heliores()
16:        {
17:            ResourceManager rm=new ResourceManager(
18:                "firstresource",this.GetType().Assembly);
19:            this.Size = new Size(400,100);
20:            this.Text=rm.GetString("WindowText");
21:
22:            Label l = new Label();
23:            l.Location = new Point(3,5);
24:            l.Size = new Size(394,90);
25:            l.Font = new Font("Tahoma",36F,FontStyle.Bold);
26:            l.Text=rm.GetString("LabelText");
27:            this.Controls.Add(l);
28:
29:        }
30:
31:        static void Main()
32:        {
33:            Application.Run(new heliores());
34:        }
35:    }
36:
37: }
```

The code in Listing 3.4.3 is a good global citizen application. It embeds no strings, and draws all of its text from the resource set provided. Line 17 of the listing shows how a resource manager is created, and lines 20 and 26 show how the named strings are retrieved from the resources in the assembly and applied to the Windows Forms elements.

To build this file and run it, you will need to use the following command line:

```
csc /out:heliores.exe /t:winexe /res:firstresource.resources heliores.cs
```

Notice how the command line option `/res:` is used to embed the compiled form of the resources created earlier.

Running the program now will produce the result shown in Figure 3.4.1.



FIGURE 3.4.1

The heliores program in operation.

3.4

Creating and Using a Satellite Assembly

The globalization example is not much use without an alternative resource set from which to draw. In the following example, we'll add a satellite assembly containing a French resource set.

NOTE

A satellite assembly is a DLL, containing code or data, that is used by an application's main assembly.

A satellite assembly, in this case, is analogous to a resource-only DLL. Satellite assemblies for localization are normally private to a particular application. They are stored in a specially named subdirectory below the main application directory. In this case, you'll need to create a directory called `fr` so that the resource manager can find it.

First, create the resource text using the same names but different values for every string that needs to be displayed. Notice that only the label text changes here. This will be explained shortly.

```
#Version Francaise.  
LabelText = Bonjour le monde!!!
```

Call this file **firstresource.fr.txt**. After this file is complete, you can create the satellite assembly using the following command lines:

```
resgen firstresource.fr.txt firstresource.fr.resources  
al /out:fr/hellores2.Resources.DLL /c:fr /embed:firstresource.fr.resources
```

The utility program invoked in the second line is the Assembly Linker (AL). AL can be used for creating assemblies of all sorts, but here we're only interested in its resource packaging abilities. Notice that the `/out:` command line option places the assembly DLL into the `fr` directory.

Now you can build the `hellores2.exe` program. It's very similar to the original but has a command line option that allows you to select the culture by typing in the culture identification string. Listing 3.4.4 shows the source.

LISTING 3.4.4 hellores2.cs: The Localized Hello World Example

```
1: namespace Sams  
2: {  
3:     using System;  
4:     using System.Drawing;  
5:     using System.Collections;  
6:     using System.ComponentModel;  
7:     using System.Globalization;  
8:     using System.Windows.Forms;  
9:     using System.Data;  
10:    using System.Resources;  
11:    using System.Threading;  
12:  
13:    class hellores: Form  
14:    {  
15:  
16:        private Label l;  
17:        private ResourceManager rm;  
18:  
19:  
20:        public hellores(string culture)  
21:        {  
22:            Thread.CurrentThread.CurrentCulture =  
23:                new CultureInfo(culture);  
24:            rm=new ResourceManager("firstresource",  
25:                this.GetType().Assembly);  
26:            this.Size = new Size(400,100);  
27:            this.Text=rm.GetString("WindowText");
```

LISTING 3.4.4 Continued

```
28:
29:         l = new Label();
30:         l.Location = new Point(3,5);
31:         l.Size = new Size(394,90);
32:         l.Font = new Font("Tahoma",36F,FontStyle.Bold);
33:         l.Text=rm.GetString("LabelText");
34:         l.Anchor = (AnchorStyles.Top |
35:                         AnchorStyles.Left |
36:                         AnchorStyles.Bottom |
37:                         AnchorStyles.Right);
38:         this.Controls.Add(l);
39:
40:     }
41:
42:     static void Main(string[] args)
43:     {
44:         string culture = "";
45:         if(args.Length == 1)
46:             culture = args[0];
47:         Application.Run(new hellores(culture));
48:     }
49: }
50:
51: }
```

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

There are a few simple changes to the program in Listing 3.4.4. Line 22 forces a setting for the current user interface culture based on the input from the command line. The properties of the label are modified on lines 34–37 so that the label changes size with the form. In this example, you will need to resize the form to see the whole label text. This shows another aspect of internationalization that you have to contend with, the possibility that physical resource sizes might be different across cultures. It is possible to store the size and placement information of your resources in the satellite assembly also. Visual Studio does this for you, and we'll look at that in a moment. Otherwise, the use of the string resources is identical.

The application can be compiled with the following command line:

```
csc /out:hellores2.exe /t:winexe /res:firstresource.resources hellores2.cs
```

Now the program can be run and an `fr` culture selected by invoking it as follows:

```
hellores2 fr
```

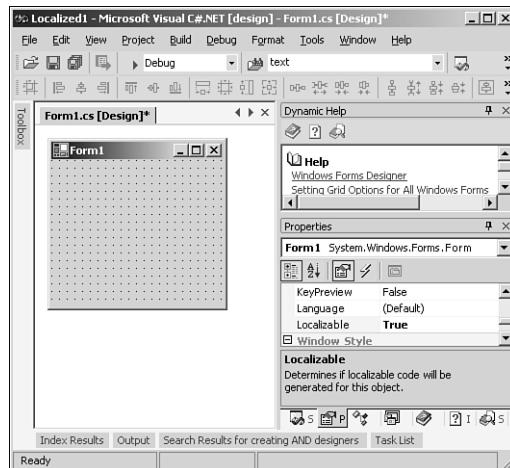
In this circumstance, you'll see the application displaying the French resource string, as shown in Figure 3.4.2.

**FIGURE 3.4.2***The French connection to resources.*

There was a reason for only modifying one of the resources in the French satellite assembly. This illustrates that the resource manager will use strings and other resources from the correct locale or culture if they exist, but it will fall back to the default information in the main assembly if they do not. The "WindowText" string is not in the satellite assembly, so the default was used.

Using Visual Studio.NET for Internationalization

Visual Studio.NET will manage resources for you and help you to create localized forms and components. Figure 3.4.3 shows the Localizable property being set for a form. Setting this property to true will cause the design time environment to store all the relevant information in the resource file for you.

**FIGURE 3.4.3***Setting the Localizable property.*

The `InitializeComponent` method for `Form1` will now contain code to initialize a resource manager and retrieve any information that might change as the application is transferred between locales from the resources.

As you saw with the handmade example shown in Figure 3.4.2, the physical extents of text strings could change as they are translated, so as well as the text itself, you will find position and size information stored in the resources. `InitializeComponent` method for `Form1` is shown in Listing 3.4.5.

LISTING 3.4.5 Form1.cs: InitializeComponent Method

```
1: private void InitializeComponent()
2: {
3:     System.Resources.ResourceManager resources =
4:         new System.Resources.ResourceManager(typeof(Form1));
5:     this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
6:     this.ClientSize = ((System.Drawing.Size)(resources.GetObject(
7:     "this.ClientSize")));
8:     this.Text = resources.GetString("this.Text");
9: }
```

You can see that the IDE has created a new resource manager on line 3, and lines 6 and 7 retrieve text and client size information from the resources.

When a form is made localizable in this way, all the components you put on the form will also save their text, size, and position information to a resource file. Listing 3.4.4 shows the `InitializeComponent` method after the addition of a label to `Form1`.

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

LISTING 3.4.4 Form1.cs: InitializeComponent after Adding a Label

```
1: private void InitializeComponent()
2: {
3:     System.Resources.ResourceManager resources =
4:         new System.Resources.ResourceManager(typeof(Form1));
5:     this.label1 = new System.Windows.Forms.Label();
6:     this.label1.Location =
7:         ((System.Drawing.Point)(resources.GetObject("label1.Location")));
8:     this.label1.Size =
9:         ((System.Drawing.Size)(resources.GetObject("label1.Size")));
10:    this.label1.TabIndex =
11:        ((int)(resources.GetObject("label1.TabIndex")));
12:    this.label1.Text = resources.GetString("label1.Text");
13:    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
14:    this.ClientSize =
15:        ((System.Drawing.Size)(resources.GetObject("this.ClientSize")));
16:    this.Controls.AddRange(new System.Windows.Forms.Control[]{this.label1});
17:    this.Text = resources.GetString("this.Text");
18: }
```

Now the label child component of the form stores its location, size, tab-index, and text in the resources.

Image Resources

So far, we have dealt with string resources but, like Win32 programs, .NET applications store images in resources for icons, backgrounds, and other things. Ultimately, the images get stored in the compiled .resource form, but, when you create them in Visual Studio or by hand, they are converted to the XML-based .resx form. Obviously, editing an image file as text in XML form is going to be no fun at all. The prescribed method for placing images in the resources is with the tools, such as VS.NET, provided by Microsoft.

A component, for example, might have an image as a background. You would place the component on the form and edit the component's background image property. The editor will allow you to select an image that is then placed into the resource for you. Figure 3.4.4 shows an image used on a form in this way.

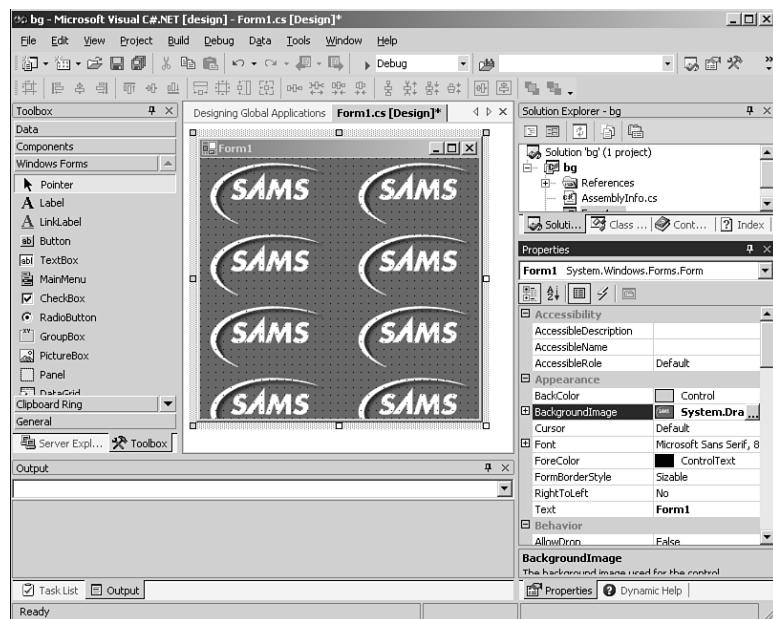


FIGURE 3.4.4

Placing a background image on a form.

Using Image Lists

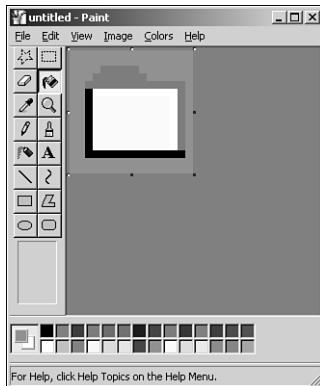
Many components require images for their user interface. The `ListView` and `TreeView` classes can use images to enhance their appearances. Image lists are collections of bitmap images held in a single file. These images can have a transparent background keyed from a chosen color so that only the chosen portion of the image is shown.

The demo code for this section will be created as we go. Using Visual Studio, create a new C# application and call it `imagelistdemo`. Follow along for the rest of the chapter to complete the job.

To create an image list with Visual Studio, you should first drag an `ImageList` component from the toolbox onto the form. Now create some images. MS Paint is suitable for this task, just remember to save the images into a common directory. Figure 3.4.5 shows a 16×16 icon being edited in MS Paint.

NOTE

For the purposes of this example, we have prepared four small icons, `icon1.bmp` through `icon4.bmp`, available from the Web site.



3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

FIGURE 3.4.5
Editing an icon image.

The background of the icon is filled with magenta. This is a color that does not appear anywhere on the actual icon image, so it is safe to use it as the transparency key. To add a collection of images to the `ImageList` object, you can select the `Images` collection in the object properties and click the Browse button. This brings up a dialog, as shown in Figure 3.4.6, that

can be used to add as many images as you need to the list. Note that it is important to keep all images in an image list the same size—in this instance, 16×16.

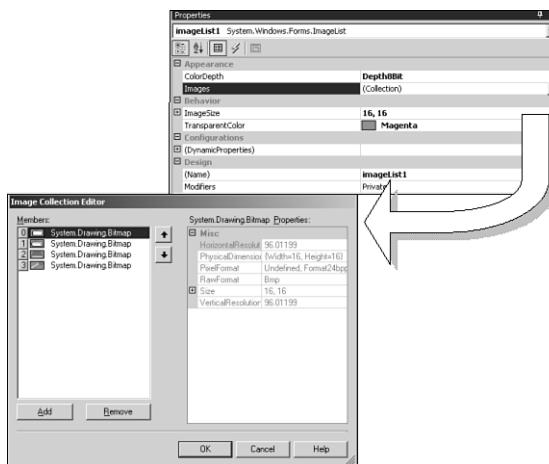


FIGURE 3.4.6

Adding images to an image list.

You can see from the image collection editor dialog that the four images are numbered from 0–3, and each has properties that can be viewed within the editor.

Now, to use these images in your `ListView` object, from the toolbox, drag a `ListView` object onto the form. For the moment, we'll simply get the `ListView` to display small icons. So select the `listView1` object and show its properties. The property called `SmallImageList` can now be set to use the `imageList1` that we just created.

Let's take a sneak peek at the `InitializeComponent` code (see Listing 3.4.5) right away to see how these tasks have affected the code.

LISTING 3.4.5 Form1.cs: InitializeComponent

```

1: private void InitializeComponent()
2: {
3:     this.components = new System.ComponentModel.Container();
4:     System.Resources.ResourceManager resources =
5:         new System.Resources.ResourceManager(typeof(Form1));
6:     this.listView1 = new System.Windows.Forms.ListView();
7:     this.imageList1 = new System.Windows.Forms.ImageList(this.components);
8:     this.SuspendLayout();
9:     // 
10:    listView1

```

LISTING 3.4.5 Continued

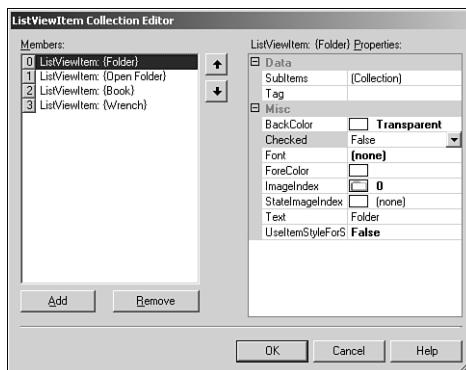
```
11:  //
12:  this.listView1.Location = new System.Drawing.Point(8, 8);
13:  this.listView1.Name = "listView1";
14:  this.listView1.Size = new System.Drawing.Size(272, 200);
15:  this.listView1.SmallImageList = this.imageList1;
16:  this.listView1.TabIndex = 0;
17:  //
18:  // imageList1
19:  //
20:  this.imageList1.ColorDepth = System.Windows.Forms.ColorDepth.Depth8Bit;
21:  this.imageList1.ImageSize = new System.Drawing.Size(16, 16);
22:  this.imageList1.ImageStream ((System.Windows.Forms.ImageListStreamer)
➥(resources.GetObject("imageList1.ImageStream")));
23:  this.imageList1.TransparentColor = System.Drawing.Color.Magenta;
24:  //
25:  // Form1
26:  //
27:  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
28:  this.ClientSize = new System.Drawing.Size(292, 273);
29:  this.Controls.AddRange(new System.Windows.Forms.Control[] {
30:    this.listView1});
31:  this.Name = "Form1";
32:  this.Text = "Form1";
33:  this.ResumeLayout(false);
34: }
```

Lines 4 and 5 created the `ResourceManager` for the form. Lines 12–16 create the `ListView` and sets up the `SmallImageList` property to use the `imageList1` component. Lines 20–24 initialize the image list by getting the image list streamer, setting the transparent color and the image size.

To use the resources in the `ListView`, select the properties for the `ListView` object, select the `Items` collection property, and click the button marked with ellipsis (...). This brings up the dialog shown in Figure 3.4.7 that lets you add and edit items in the list box.

NOTE

Note that your programs can also accomplish adding items to components dynamically. Visual Studio provides an additional RAD way of doing things.

**FIGURE 3.4.7**

Using resource-based images in *ListViewItem* objects.

Each of the items added to the tree can be given some text and an image to display. The *ListView* also provides events similar to those seen in the tree view demonstration from Chapter 3.2, “User Interface Components,” when list items are selected, edited, and so on. Handling these events allows you to interact with the list.

Let’s take a look now at the final *InitializeComponent* listing for the example shown (see Listing 3.4.6).

LISTING 3.4.6 Form1.cs: *InitializeComponent*

```
1: private void InitializeComponent()
2: {
3:     this.components = new System.ComponentModel.Container();
4:     System.Resources.ResourceManager resources = new
5:         System.Resources.ResourceManager(typeof(Form1));
6:     System.Windows.Forms.ListViewItem listViewItem1 = new
7:         System.Windows.Forms.ListViewItem("Folder", 0);
8:     System.Windows.Forms.ListViewItem listViewItem2 = new
9:         System.Windows.Forms.ListViewItem("Open Folder", 1);
10:    System.Windows.Forms.ListViewItem listViewItem3 = new
11:        System.Windows.Forms.ListViewItem("Book", 2);
12:    System.Windows.Forms.ListViewItem listViewItem4 = new
13:        System.Windows.Forms.ListViewItem("Wrench", 3);
14:    this.imageList1 = new System.Windows.Forms.ImageList(this.components);
15:    this.listView1 = new System.Windows.Forms.ListView();
16:    this.SuspendLayout();
17:    //
18:    // imageList1
19:    //
```

LISTING 3.4.6 Continued

```
20:     this.imageList1.ColorDepth = System.Windows.Forms.ColorDepth.Depth8Bit;
21:     this.imageList1.ImageSize = new System.Drawing.Size(16, 16);
22:     this.imageList1.ImageStream = ((System.Windows.Forms.ImageListStreamer)
23:         ➔ (resources.GetObject("imageList1.ImageStream")));
24:     this.imageList1.TransparentColor = System.Drawing.Color.Magenta;
25:     //
26:     // listView1
27:     //
28:     listViewItem1.UseItemStyleForSubItems = false;
29:     listViewItem2.UseItemStyleForSubItems = false;
30:     listViewItem3.UseItemStyleForSubItems = false;
31:     listViewItem4.UseItemStyleForSubItems = false;
32:     this.listView1.Items.AddRange(new System.Windows.Forms.ListViewItem[] {
33:         listViewItem1,
34:         listViewItem2,
35:         listViewItem3,
36:         listViewItem4});
37:     this.listView1.Location = new System.Drawing.Point(8, 8);
38:     this.listView1.Name = "listView1";
39:     this.listView1.Size = new System.Drawing.Size(272, 200);
40:     this.listView1.SmallImageList = this.imageList1;
41:     this.listView1.TabIndex = 0;
42:     this.listView1.View = System.Windows.Forms.View.SmallIcon;
43:     //
44:     // Form1
45:     //
46:     this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
47:     this.ClientSize = new System.Drawing.Size(292, 273);
48:     this.Controls.AddRange(new System.Windows.Forms.Control[]
49:         {➔this.listView1});
50:     this.Name = "Form1";
51:     this.Text = "Form1";
52:     this.ResumeLayout(false);
53: }
```

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

Now, lines 6–13 create the `ListViewItem` objects and set the text and image offset within the `ImageList` to use. Lines 32–36 add the items to the `ListView`.

Programmatic Access to Resources

In addition to creating resources with Visual Studio or the other tools provided, you can create, manage, and use resources easily through code. An example of this application of resources would be to store some custom data for your application (for example, window sizes and positions) to be retrieved when the program was run again.

Reading and writing resources are performed with the `ResourceReader` and `ResourceWriter` classes. These objects let you deal with resources stored in streams or in files.

In the example that follows, we have prepared a simple Windows Forms application that displays a red and yellow ball on the form's surface. You can pick these balls up with the mouse and move them about. When the application is closed, it creates and writes a resource called `ball_locations.resources`. This resource stores the positions onscreen of the two balls so that the next time it is loaded, the application replaces the balls where you left them.

As a bonus, this application shows some simple mouse handling and the use of the `ImageList` to draw images on the form surface.

LISTING 3.4.7 `Resourcerw.cs`: The Resource Read/Write Application

```
1: using System;
2: using System.Drawing;
3: using System.Collections;
4: using System.ComponentModel;
5: using System.Windows.Forms;
6: using System.Data;
7: using System.Resources;
8: using System.Globalization;
9:
10:
11:
12: namespace resourcerw
13: {
14:     /// <summary>
15:     /// Summary description for Form1.
16:     /// </summary>
17:     public class Form1 : System.Windows.Forms.Form
18:     {
19:         private System.Windows.Forms.ImageList imageList1;
20:         private System.ComponentModel.IContainer components;
21:         private System.Drawing.Point[] BallLocations;
22:
23:         private bool _mousedown;
24:         private int _grabbed;
25:
26:         public Form1()
27:         {
28:             //
29:             // Required for Windows Form Designer support
30:             //
31:             InitializeComponent();
32:         }
```

LISTING 3.4.7 Continued

```
33:      //  
34:      // TODO: Add any constructor code after InitializeComponent call  
35:      //  
36:  
37:      this.BallLocations=new System.Drawing.Point[2];  
38:  
39:  }  
40:  
41:  /// <summary>  
42:  /// Clean up any resources being used.  
43:  /// </summary>  
44:  protected override void Dispose( bool disposing )  
45:  {  
46:      if( disposing )  
47:      {  
48:          if (components != null)  
49:          {  
50:              components.Dispose();  
51:          }  
52:      }  
53:      base.Dispose( disposing );  
54:  }  
55:  
56: #region Windows Form Designer generated code  
57:  /// <summary>  
58:  /// Required method for Designer support - do not modify  
59:  /// the contents of this method with the code editor.  
60:  /// </summary>  
61:  private void InitializeComponent()  
62:  {  
63:      this.components = new System.ComponentModel.Container();  
64:      System.Resources.ResourceManager resources =  
65:          new System.Resources.ResourceManager(typeof(Form1));  
66:      this.imageList1 =  
67:          new System.Windows.Forms.ImageList(this.components);  
68:      //  
69:      // imageList1  
70:      //  
71:      this.imageList1.ColorDepth =  
72:          System.Windows.Forms.ColorDepth.Depth8Bit;  
73:      this.imageList1.ImageSize = new System.Drawing.Size(64, 64);  
74:      this.imageList1.ImageStream =  
75:          ((System.Windows.Forms.ImageListStreamer)  
76:             (resources.GetObject("imageList1.ImageStream")));  
77:      this.imageList1.TransparentColor = System.Drawing.Color.Magenta;
```

LISTING 3.4.7 Continued

```
78:      //  
79:      // Form1  
80:      //  
81:      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);  
82:      this.BackColor = System.Drawing.Color.Green;  
83:      this.ClientSize = new System.Drawing.Size(376, 301);  
84:      this.Name = "Form1";  
85:      this.Text = "Resource read-write";  
86:      this.MouseDown +=  
87:          new System.Windows.Forms.MouseEventHandler(this.Form1_MouseDown);  
88:      this.Closing +=  
89:          new System.ComponentModel.CancelEventHandler(this.Form1_Closing);  
90:      this.Load +=  
91:          new System.EventHandler(this.Form1_Load);  
92:      this.MouseUp +=  
93:          new System.Windows.Forms.MouseEventHandler(this.Form1_MouseUp);  
94:      this.Paint +=  
95:          new System.Windows.Forms.PaintEventHandler(this.Form1_Paint);  
96:      thisMouseMove +=  
97:          new System.Windows.Forms.MouseEventHandler(this.Form1_MouseMove);  
98:  
99:    }  
100:   #endregion  
101:  
102:  /// <summary>  
103:  /// The main entry point for the application.  
104:  /// </summary>  
105:  [STAThread]  
106:  static void Main()  
107:  {  
108:    Application.Run(new Form1());  
109:  }  
110:  
111:  
112:  private void Form1_MouseMove(object sender,  
113:                                System.Windows.Forms.MouseEventArgs e)  
114:  {  
115:    if(_mousedown && (_grabbed!=-1))  
116:    {  
117:      if(e.X>31 && e.Y>31 && e.X<(this.Size.Width-32) &&  
118:         e.Y<(this.Size.Height-32))  
119:      {  
120:        BallLocations[_grabbed].X=e.X;  
121:        BallLocations[_grabbed].Y=e.Y;  
122:      }
```

LISTING 3.4.7 Continued

```
123:         this.Invalidate();
124:     }
125: }
126: }
127:
128: private void Form1_MouseUp(object sender,
129:     System.Windows.Forms.MouseEventArgs e)
130: {
131:     _mousedown=false;
132: }
133:
134: private void Form1_MouseDown(object sender,
135:     System.Windows.Forms.MouseEventArgs e)
136: {
137:     _mousedown=true;
138:     int index=0;
139:     _grabbed=-1;
140:     foreach(Point p in this.BallLocations)
141:     {
142:         if(Math.Abs(e.X-p.X)<32 && Math.Abs(e.Y-p.Y)<32)
143:             _grabbed=index;
144:             index++;
145:     }
146: }
147:
148: private void Form1_Paint(object sender,
149:     System.Windows.Forms.PaintEventArgs e)
150: {
151:     int index=0;
152:     foreach(Point p in this.BallLocations)
153:     {
154:         this.imageList1.Draw(e.Graphics,p.X-32,p.Y-32,64,64,index++);
155:     }
156: }
157:
158: private void Form1_Load(object sender, System.EventArgs e)
159: {
160:
161:     ResourceSet rs;
162:
163:     try
164:     {
165:         rs = new ResourceSet("ball_locations.resources");
166:         BallLocations[0] = (Point)rs.GetObject("RedBall",false);
167:         BallLocations[1] = (Point)rs.GetObject("YellowBall",false);
```

LISTING 3.4.7 Continued

```
168:         rs.Close();
169:     }
170:     catch(System.Exception)
171:     {
172:         // Any old exception will do, probably file not yet created...
173:         BallLocations[0]=new Point(100,100);
174:         BallLocations[1]=new Point(200,200);
175:     }
176:
177: }
178:
179: private void Form1_Closing(object sender,
180:                             System.ComponentModel.CancelEventArgs e)
181: {
182:     // Write the ball positions to the custom resource
183:     // note you can just write objects.
184:     ResourceWriter rw = new ResourceWriter("ball_locations.resources");
185:     rw.AddResource("RedBall",this.BallLocations[0]);
186:     rw.AddResource("YellowBall",this.BallLocations[1]);
187:     rw.Generate();
188:     rw.Close();
189: }
190:
191:
192: }
193: }
```

Lines 61–99 show the `InitializeComponent` method required by Visual Studio. Lines 63–65 open the image resources used by the `ImageList` object, and lines 66–77 initialize the `ImageList` by loading in the image stream and setting the parameters for size and transparent color. Lines 86–97 add the mouse and paint other handlers.

Now for the event handlers themselves. The two important ones are the form loading and form closing events. Lines 158–175 are the form load event handler. This attempts to open a resource and get the ball positions from it. If this fails for any reason, an exception handler simply assumes that the file doesn't exist and initializes both balls to their default positions. Lines 179–188 are invoked when the form is ready to close. Here, the code creates a resource file and stores all the ball positions in it. The next time the program is run, this file will exist and the load event handler will successfully find the stored positions.

The `MouseUp` handler (lines 128–132) simply resets a semaphore that signifies no dragging is taking place. The `MouseDown` handler (lines 134–146) decides which of the two balls, if any, the mouse is in, and flags the correct one to grab.

The `MouseMove` handler checks to see if a ball is grabbed and if the mouse is inside the client area, so that you cannot drag a ball off the screen. All being correct, it updates the position of the currently active ball. This method is found on lines 112–126.

Lastly, on lines 148–156 the `Paint` handler paints the balls on the surface of the form.

Reading and Writing RESX XML Files

Resource files, as we mentioned at the beginning of this chapter, come in several flavors—`.resources` files and `.resx` files. `Resx` files are the raw, uncompiled XML used as a basic source form for resources. In the previous example, we showed the `ResourceReader` and `ResourceWriter` classes. There are corresponding `ResXResourceReader` and `ResXResourceWriter` classes that can be used to maintain the XML forms of the resources.

In our last example for this chapter, we've created a simple application that lets you load chosen bitmap, text, and icon files and convert these to a `.RESX` resource file (see Listing 3.4.8). The complexity of the task and the simplicity of this application show off some of the raw power of the Windows Forms architecture.

LISTING 3.4.8 ResXUtilForm.cs: The RESX Writer Utility

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

```
1: using System;
2: using System.Drawing;
3: using System.Collections;
4: using System.ComponentModel;
5: using System.Windows.Forms;
6: using System.Data;
7: using System.IO;
8:
9: /*
10:  * NOTES: A bug in an early version of the .NET framework caused
11:  * the program to raise an exception when the resource name is
12:  * edited. In this file there are two sections of commented code
13:  * that are designed to work around this problem
14:  * If this code fails, simply uncomment the sections marked
15:  * BUGFIXER and re-compile
16:  *
17: */
18:
19: namespace ResxUtil
20: {
21:     /// <summary>
22:     /// Summary description for Form1.
23:     /// </summary>
24:     public class ResXUtilForm : System.Windows.Forms.Form
```

LISTING 3.4.8 Continued

```
25:  {
26:      private System.Windows.Forms.ListView listView1;
27:      private System.Windows.Forms.Button button1;
28:      private System.Windows.Forms.Button button2;
29:      private System.Windows.Forms.ImageList imageList1;
30:      private System.Windows.Forms.ColumnHeader columnHeader2;
31:      private System.Windows.Forms.ColumnHeader columnHeader1;
32:      private System.ComponentModel.IContainer components;
33:
34:      public ResXUtilForm()
35:      {
36:          //
37:          // Required for Windows Form Designer support
38:          //
39:          InitializeComponent();
40:
41:          //BUGFIXER uncomment the line below
42:          //this.listView1.LabelEdit=false;
43:      }
44:
45:      /// <summary>
46:      /// Clean up any resources being used.
47:      /// </summary>
48:      protected override void Dispose( bool disposing )
49:      {
50:          if( disposing )
51:          {
52:              if (components != null)
53:              {
54:                  components.Dispose();
55:              }
56:          }
57:          base.Dispose( disposing );
58:      }
59:
60:      #region Windows Form Designer generated code
61:      /// <summary>
62:      /// Required method for Designer support - do not modify
63:      /// the contents of this method with the code editor.
64:      /// </summary>
65:      private void InitializeComponent()
66:      {
67:          this.components = new System.ComponentModel.Container();
68:          System.Resources.ResourceManager resources =
➥new System.Resources.ResourceManager(typeof(ResXUtilForm));
```

LISTING 3.4.8 Continued

```
69:     this.columnHeader2 = new System.Windows.Forms.ColumnHeader();
70:     this.columnHeader1 = new System.Windows.Forms.ColumnHeader();
71:     this.imageList1 =
    ➔new System.Windows.Forms.ImageList(this.components);
72:     this.listView1 = new System.Windows.Forms.ListView();
73:     this.button1 = new System.Windows.Forms.Button();
74:     this.button2 = new System.Windows.Forms.Button();
75:     this.SuspendLayout();
76:     //
77:     // columnHeader2
78:     //
79:     this.columnHeader2.Text = "File name";
80:     this.columnHeader2.Width = 545;
81:     //
82:     // columnHeader1
83:     //
84:     this.columnHeader1.Text = "Resource name";
85:     this.columnHeader1.Width = 220;
86:     //
87:     // imageList1
88:     //
89:     this.imageList1.ColorDepth =
    ➔System.Windows.Forms.ColorDepth.Depth8Bit;
90:     this.imageList1.ImageSize = new System.Drawing.Size(16, 16);
91:     this.imageList1.ImageStream =
    ➔ ((System.Windows.Forms.ImageListStreamer)
    ➔(resources.GetObject("imageList1.ImageStream")));
92:     this.imageList1.TransparentColor = System.Drawing.Color.Magenta;
93:     //
94:     // listView1
95:     //
96:     this.listView1.Anchor = ((System.Windows.Forms.AnchorStyles.Top |
    ➔System.Windows.Forms.AnchorStyles.Left)
97:         | System.Windows.Forms.AnchorStyles.Right);
98:     this.listView1.Columns.AddRange(
    ➔new System.Windows.Forms.ColumnHeader[] {
99:             this.columnHeader1,
100:             this.columnHeader2});
101:    this.listView1.FullRowSelect = true;
102:    this.listView1.Location = new System.Drawing.Point(8, 8);
103:    this.listView1.Name = "listView1";
104:    this.listView1.Size = new System.Drawing.Size(792, 304);
105:    this.listView1.SmallImageList = this.imageList1;
106:    this.listView1.TabIndex = 0;
107:    this.listView1.View = System.Windows.Forms.View.Details;
```

LISTING 3.4.8 Continued

```
108:     this.listView1.KeyDown +=
  ↪new System.Windows.Forms.KeyEventHandler(this.listView1_KeyDown);
109:     this.listView1.LabelEdit=true;
110:     //
111:     // button1
112:     //
113:     this.button1.Location = new System.Drawing.Point(696, 328);
114:     this.button1.Name = "button1";
115:     this.button1.Size = new System.Drawing.Size(104, 32);
116:     this.button1.TabIndex = 1;
117:     this.button1.Text = "Save";
118:     this.button1.Click += new System.EventHandler(this.button1_Click);
119:     //
120:     // button2
121:     //
122:     this.button2.Location = new System.Drawing.Point(8, 328);
123:     this.button2.Name = "button2";
124:     this.button2.Size = new System.Drawing.Size(88, 32);
125:     this.button2.TabIndex = 2;
126:     this.button2.Text = "Add... ";
127:     this.button2.Click += new System.EventHandler(this.button2_Click);
128:     //
129:     // ResXUtilForm
130:     //
131:     this.AllowDrop = true;
132:     this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
133:     this.ClientSize = new System.Drawing.Size(816, 373);
134:     this.Controls.AddRange(new System.Windows.Forms.Control[] {
135:                                     this.button2,
136:                                     this.button1,
137:                                     this.listView1});
138:     this.FormBorderStyle =
  ↪System.Windows.Forms.FormBorderStyle.FixedDialog;
139:     this.Name = "ResXUtilForm";
140:     this.Text = "ResX Utility";
141:     this.ResumeLayout(false);
142:
143: }
144: #endregion
145:
146: /// <summary>
147: /// The main entry point for the application.
148: /// </summary>
149: [STAThread]
150: static void Main()
```

LISTING 3.4.8 Continued

```
151:    {
152:        Application.Run(new ResXUtilForm());
153:    }
154:
155:    private void button2_Click(object sender, System.EventArgs e)
156:    {
157:        int icon=-1;
158:        OpenFileDialog dlg=new OpenFileDialog();
159:        if(dlg.ShowDialog()==DialogResult.OK)
160:        {
161:            string s=dlg.FileName;
162:            string[] d=s.Split(new char[]{'.',''});
163:            string resname="Unidentified";
164:
165:            if(d.Length<2)
166:                return;
167:
168:            switch(d[d.Length-1].ToLower())
169:            {
170:                case "bmp":
171:                    icon=0;
172:                    break;
173:                case "txt":
174:                    icon=1;
175:                    break;
176:                case "ico":
177:                    icon=2;
178:                    break;
179:            }
180:
181:
182:            //BUGFIXER remove comment block tags from the
183:            //code below.
184:            /*Form f=new Form();
185:            f.Size=new Size(350,110);
186:            f.Text="Resource name";
187:            f.FormBorderStyle=FormBorderStyle.FixedDialog;
188:            Button ok=new Button();
189:            ok.DialogResult=DialogResult.OK;
190:            ok.Location=new Point(5,32);
191:            ok.Size=new Size(75,22);
192:            ok.Text="Ok";
193:            f.Controls.Add(ok);
194:            Button can=new Button();
```

LISTING 3.4.8 Continued

```
195:         can.DialogResult=DialogResult.Cancel;
196:         can.Location=new Point(260,32);
197:         can.Size=new Size(75,22);
198:         can.Text="Cancel";
199:         f.Controls.Add(can);
200:         TextBox tb=new TextBox();
201:         tb.Location=new Point(5,5);
202:         tb.Size=new Size(330,25);
203:         tb.Text="Resource"+
204:             ((object)(this.listView1.Items.Count+1)).ToString();
205:         f.Controls.Add(tb);
206:         if(f.ShowDialog()==DialogResult.Cancel || tb.Text.Trim() == "")
207:             return;
208:         resname=tb.Text.Trim();
209:         */
210:
211:
212:         if(icon>=0)
213:         {
214:             ListViewItem item=new ListViewItem(resname,icon);
215:             item.SubItems.Add(s);
216:             this.listView1.Items.Add(item);
217:         }
218:     }
219: }
220:
221: private void button1_Click(object sender, System.EventArgs e)
222: {
223:     if(this.listView1.Items.Count>0)
224:     {
225:         //Check that all resources have a name
226:         foreach(ListViewItem item in this.listView1.Items)
227:         {
228:             if(item.Text=="Undefined")
229:             {
230:                 MessageBox.Show("All resources must have names");
231:                 return;
232:             }
233:         }
234:
235:         //Fetch the filename for the resx file
236:         SaveFileDialog dlg=new SaveFileDialog();
237:         dlg.Title="Save .resx file";
238:         dlg.Filter="XML resource files|.resx";
```

LISTING 3.4.8 Continued

```
239:         if(dlg.ShowDialog()==DialogResult.Cancel)
240:             return;
241:
242:         //Create a resource writer
243:         System.Resources.ResXResourceWriter rw=
244:             new System.Resources.ResXResourceWriter(dlg.FileName);
245:
246:         //Step through the items on the list
247:         foreach(ListViewItem item in this.listView1.Items)
248:         {
249:             switch(item.ImageIndex)
250:             {
251:                 case 0: // bitmap images
252:                     goto case 2;
253:                 case 1: // text files
254:                     System.IO.StreamReader sr=
255:                         new System.IO.StreamReader(
256:                             item.SubItems[1].Text);
257:                     string inputline;
258:                     do
259:                     {
260:                         inputline = sr.ReadLine();
261:                         if(inputline != null)
262:                         {
263:                             string[] splitline=
264:                                 inputline.Split(new char[]{ '=' },2);
265:                             if(splitline.Length==2)
266:                             {
267:                                 rw.AddResource(
268:                                     item.Text.TrimEnd().TrimStart()+
269:                                     "."+
270:                                     splitline[0].TrimEnd().TrimStart(),
271:                                     splitline[1]);
272:                             }
273:                         }
274:                     }
275:                     while(inputline!=null);
276:                     break;
277:                 case 2: // icons
278:                     System.Drawing.Image im=
279:                         System.Drawing.Image.FromFile(
280:                             item.SubItems[1].Text);
281:                     rw.AddResource(item.Text,im);
282:                     break;
```

LISTING 3.4.8 Continued

```
283:         }
284:     }
285:
286:     rw.Generate();
287:     rw.Close();
288:   }
289: }
290:
291:
292: private void listView1_KeyDown(object sender,
293:                               System.Windows.Forms.KeyEventArgs e)
294: {
295:   if(this.listView1.SelectedIndices.Count==0)
296:     return;
297:   if(e.KeyCode.Equals(Keys.Delete))
298:     this.listView1.Items.RemoveAt(
299:       this.listView1.SelectedIndices[0]);
300:   }
301: }
302:
303: }
```

NOTE

Be aware that the style of layout in this and other listings has been modified to fit an 80 column format. In particular, the `InitializeComponent` method has been modified from its original form. If you download the file from the Sams Web site, it might differ slightly in layout, but it will be functionally identical.

By now, you should be able to go through the `InitializeComponent` and understand what's happening, so we'll just pick out highlights of the application.

Lines 79–80 create a set of column headers used when the `ListView` is in Detail mode to provide headers for each column of information. We have chosen to display the item icon from an `ImageList` plus the name of the resource, which you must choose, along with the full filename of the source resource.

Lines 89–92 set up the `ImageList`, bringing in the image stream and setting up sizes and transparent color and so on.

Lines 98–100 add the column headers to the list view control. Lines 93–104 set the properties of the `ListView` to enable row selection (so you can delete an object from the list), label editing (which lets you name your resource, locations, and sizes for the control), and finally, the `KeyDown` handler (which only responds to the Delete key).

Lines 110–128 are the buttons. One adds a resource, the other saves the RESX file.

Lines 155–219 handle the Add button. This allows you to browse files (lines 154 and 155) for bitmaps, text, or icons. After you find a resource candidate, lines 168–179 decide whether it's a file we care about and select the correct icon. Finally, lines 212–217 use the icon selection to create a `ListViewItem` that has a label (at first “Undefined”) and a sub-item that populates the second column of the list control with the filename.

The Save button on lines 221–284 does a quick check to see if all the resources have been named (lines 226–233) and then gets a filename for the target file (lines 236–240). Lines 243–244 make a new `ResXResourceWriter` and then pass it into a `foreach` loop that goes through item by item and creates the resources.

The switch statement (line 249) has three cases. Cases 0 and 2 are handled the same way because, for the XML format, there is no distinction between a bitmap and an icon. Case 1 (lines 254–276) reads in the text file one line at a time and separates the lines into the name-value pairs used to create each text resource. The name is prefixed with the name you chose for the text file in the listbox. Note that this demonstration only understands text files with the "`<name string>=<value string>`" syntax. Other files are likely to give errors. After processing is done, lines 286 and 287 write and close the RESX file.

Finally, lines 292–300 handle the Delete key presses and remove items from the list if you want.

3.4

WINDOWS FORMS
EXAMPLE
APPLICATION

A Note about Bugs

Line 42 and the block of code on lines 184–208 are a contingency plan that solves a potential problem with an early version of the .NET framework. If you edit the resource name and the program crashes, uncomment the two sections of code in the lines marked "BUGFIXER" and recompile the program. It will work in a slightly different way, and it will not crash.

Notice also how the `bugfix` code creates a dialog box on-the-fly for you.

NOTE

For applications built with Visual Studio, the complete solution is included as a zip archive on the Sams Web site (<http://www.samspublishing.com>).

Summary

In this chapter, you have seen how to create text resources by hand, how to internationalize applications using satellite assemblies, how to use Visual Studio to create applications with image and other resources, and how to dynamically create and use resource files in your programs. In the next chapter, we'll be looking at the powerful graphics options available to you with GDI+.

IN THIS CHAPTER

- The Basic Principles of GDI+ 332
- The Graphics Object 332
- Graphics Coordinates 333
- Drawing Lines and Simple Shapes 333
- Using Gradient Pens and Brushes 336
- Textured Pens and Brushes 338
- Tidying up Your Lines with Endcaps 339
- Curves and Paths 341
- The GraphicsPath Object 349
- Clipping with Paths and Regions 352
- Transformations 357
- Alpha Blending 365
- Alpha Blending of Images 367
- Other Color Space Manipulations 371

In this chapter, we'll look at the Graphical Device Interface (GDI) used by Windows Forms to create graphics. Notice the name is GDI+ not GDI.NET. This is because GDI+ is not solely a .NET Framework API. It can also be used by unmanaged code. For the purposes of this chapter however, we'll use it exclusively from Windows Forms.

The Basic Principles of GDI+

Like its predecessor GDI, GDI+ is an immediate mode graphics system. This means that as you send commands to the graphic interface, effects are seen on the device surface or in the display memory immediately. The other kind of graphics system, retained mode graphics, usually maintain hierarchies of objects that know how to paint themselves.

GDI+ uses brushes and pens to paint on the graphics surface, just like GDI did. Unlike GDI, though, you can use any graphical object on any surface at any time. GDI required you to create a graphics object, select it for use in a Device Context (DC), use the object, deselect it from the DC, and then destroy the object. If you didn't do it in that order, you could get system resource leaks that would eventually cause all the memory in your system to be allocated away and the system would need a reboot. GDI+ is altogether friendlier to use because you don't have to worry about selecting and deselecting objects in the right order.

As you might expect, the evolution of GDI+ reflects many of the recent changes in graphics card capabilities. Additions to your graphics arsenal include Alpha Blending or image transparency, antialiasing, transformations like rotation and scaling, plus many new innovations in the areas of color correction and region manipulation.

The Graphics Object

The root of all GDI+ graphics capabilities for .NET is the `Graphics` object. This is analogous to the Win32 DC and can be obtained explicitly from a window handle or is passed to your application from, say, the `OnPaint` event. To obtain a `Graphics` object in Managed C++, for example, you would write the following code:

```
HDC dc;
PAINTSTRUCT ps;
Graphics *pGraphics;

Dc=BeginPaint(hWnd,&ps)
pGraphics=new Graphics(dc);
// use the Graphics...
EndPaint(hWnd,&ps);
```

For the purposes of this chapter, we'll assume that GDI+ operations will take place in the context of a Windows Forms application and that the `Graphics` object is handed to you in an event delegate. For example, we see the `Graphics` object that's supplied in the `PaintEventArgs` in the following:

```
public void OnPaint(object sender, PaintEventArgs e)
{
    // use e.Graphics.SomeMethod(...) here
```

Graphics Coordinates

Objects are painted onto the screen using coordinates that are mapped to no less than three systems. The World Coordinate system, which is an abstract concept of space within the graphics system, the Page Coordinate system, which is another abstract concept of physical dimension on a hypothetical page, and the Device Coordinate system, which is tied to the physical properties of the screen or printer upon which the graphics are rendered. To display anything onscreen, the graphical data that you write to the GDI+ graphics object has to be transformed several times. The reason for this is to allow you to deal with graphics in a device-independent way. For example, if you place a shape or some text in World Coordinates, you might want to ensure that the resulting graphic is displayed as though it were on a standard A4 sheet of paper. The image must look as if it's on A4 both on the monitor and on any printer of any resolution. The World Coordinate is, therefore, mapped to a Page Coordinate that represents an 8 1/2" by 11" physical measurement. When the system comes to display the A4 page, the monitor might have a resolution of 72 dots per inch (DPI)—printer A has a resolution of 300 DPI and printer B has a resolution of 1200 by 400 DPI. Page space, a physical measurement, then needs to be mapped to Device Space so that, in reality, the monitor maps 8.5 inches across the page to 612 pixels on the monitor—2550 pixels on printer A and 10200 pixels on printer B.

GDI+ allows you to set the way that objects are transformed from the world to the page using the `PageUnit`, `PageUnit`, `PageScale`, and `PageScale` properties. You have no control over the transformation to device space. That all happens in the device drivers themselves. You can, however, read the device's DPI with the `DpiX` and `DpiY` properties.

We'll examine coordinate transformations in detail later in this chapter.

3.5

GDI+: THE .NET
GRAPHICS
INTERFACE

Drawing Lines and Simple Shapes

Drawing lines or the outline of a shape requires the use of a pen. Filling an area, particularly the interior of a shape, is accomplished with a brush. The simple Windows Forms example in Listing 3.5.1 shows these operations. This is a self-contained, hand-edited example, so Visual Studio.NET is not needed.

NOTE

A word about graphics object management under GDI+ for Windows Forms. The garbage collection heap, as you know by now, does memory management under .NET for you. Technically, there is no need to worry about reclaiming your objects by deleting them because the garbage collector will do it for you eventually.

This is also true for graphical operations that create pens and brushes. However, an intense graphics routing could create many thousands of pens or brushes during each draw cycle and, in circumstances such as this, the garbage collector might not be quick enough to reclaim all the objects, causing a large number of dead items to be left on the heap for a considerable period of time.

To minimize the amount of work the garbage collector does, you should, wherever possible, explicitly invoke the `Dispose()` method on pens, brushes, and other such objects. This reclaims the memory in the object and tells the garbage collector not to bother with them.

In certain examples, you will see the declaration of the object inline, for example, `DrawRectangle(new Pen(Color.Red), ...)`. This will not cause a memory leak of the pen, it simply defers reclamation of the object when it's no longer in scope. In other examples, you will see explicit use of, for example, `myPen.Dispose()`. This is the recommended method for highest memory performance.

LISTING 3.5.1 DrawLines.cs: Drawing Lines and Filling Rectangles

```
1: using System;
2: using System.Drawing;
3: using System.Windows.Forms;
4:
5: class drawlines : System.Windows.Forms.Form
6: {
7:     Timer t;
8:     bool BackgroundDirty;
9:
10:    public void TickEventHandler(object sender, EventArgs e)
11:    {
12:        Invalidate();
13:    }
14:
15:    public void OnPaint(object sender, PaintEventArgs e)
16:    {
17:        // the current graphics object for
18:        // this window is in the PaintEventArgs
19:
```

LISTING 3.5.1 Continued

```
20:      Random r=new Random();
21:
22:      Color c=Color.FromArgb(r.Next(255),r.Next(255),r.Next(255));
23:      Pen p=new Pen(c,(float)r.NextDouble()*10);
24:      e.Graphics.DrawLine(p,r.Next(this.ClientSize.Width),
25:                           r.Next(this.ClientSize.Height),
26:                           r.Next(this.ClientSize.Width),
27:                           r.Next(this.ClientSize.Height));
28:      p.Dispose();
29:  }
30:
31:  protected override void OnPaintBackground(PaintEventArgs e)
32:  {
33:      // When we resize or on the first time run
34:      // we'll paint the background, otherwise
35:      // it will be left so the lines build up
36:      if(BackgroundDirty)
37:      {
38:          BackgroundDirty = false;
39:          // We could call base.OnPaintBackground(e);
40:          // but that doesn't show off rectangle filling
41:          e.Graphics.FillRectangle(new SolidBrush(this.BackColor),
42:                                   this.ClientRectangle);
43:      }
44:
45:  }
46:
47:  public void OnSized(object sender, EventArgs e)
48:  {
49:      BackgroundDirty=true;
50:      Invalidate();
51:  }
52:
53:  public drawlines()
54:  {
55:      t=new Timer();
56:      t.Interval=300;
57:      t.Tick+=new System.EventHandler(TickEventHandler);
58:      t.Enabled=true;
59:      this.Paint+=new PaintEventHandler(OnPaint);
60:      this.SizeChanged+=new EventHandler(OnSized);
61:      this.Text="Lines and lines and lines";
62:      BackgroundDirty = true;
63:  }
```

LISTING 3.5.1 Continued

```
64:  
65:    static void Main()  
66:    {  
67:        Application.Run(new drawlines());  
68:    }  
69: };
```

Compile Listing 3.5.1 by using the following command line:

```
csc /t:winexe drawlines.cs
```

Lines 53–62 set up the application by creating a timer and by adding the `OnPaint` handler to the `Paint` event list. It also adds an event handler for the `SizeChanged` event, so that we can detect when a new background needs painting.

Lines 10–13 are a simple `Tick` event handler that just invalidates the window. And lines 15–29 are the paint handler that creates a pen and draws a line between two random coordinates onscreen.

Lines 31–43 paint the background for us. The reason that it's done this way is so that the invalidation of the screen doesn't paint over the lines already drawn. You can take this out if you like to see what happens. It illustrates how a brush fills a rectangle.

Using Gradient Pens and Brushes

Pens and brushes have come a long way in a short time. GDI+ allows you to have lines and filled areas that show a gradient or sweep of colors. Modifying the code in Listing 3.5.1 will allow us to use the gradient pens or gradient fills instead of solid lines or colors.

To fill the background of the window is simple enough, we just need to specify a gradient fill brush. The `LinearGradientBrush` object is a member of the `System.Drawing.Drawing2D` namespace. Drawing a gradient fill on a line is very simple, because one of the overloads allows you to specify two `Point` objects, each at opposite corners of a rectangle.

Listing 3.5.2 is a selective listing of the two sections you need to change in the example from Listing 3.5.1. It contains only the modified routines `OnPaint` and `OnPaintBackground`. Remember, though, to add the declaration

```
Using System.Drawing.Drawing2D;
```

to your source also.

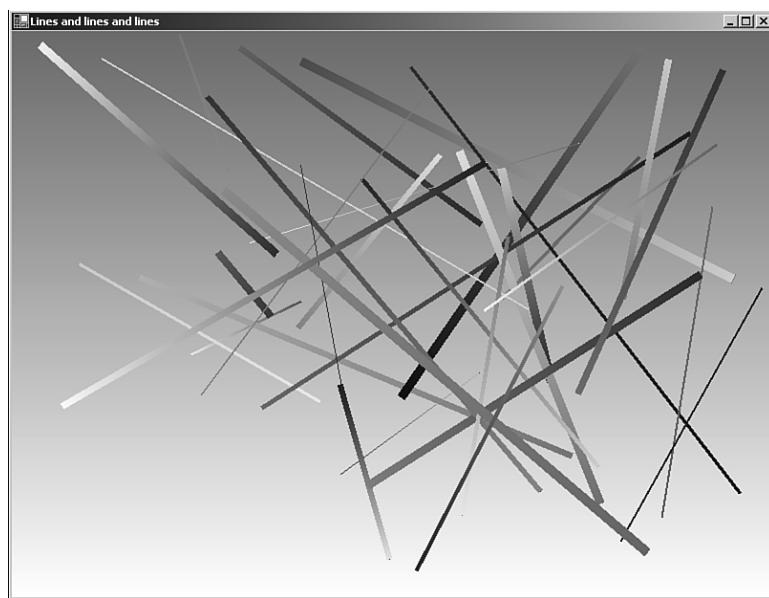
LISTING 3.5.2 DrawLines2.cs: The Modified Line and Fill Code for DrawLines.cs

```
1: 
2: public void OnPaint(object sender, PaintEventArgs e)
3: {
4:     // the current graphics object for
5:     // this window is in the PaintEventArgs
6: 
7:     Random r=new Random();
8: 
9:     Color cA=Color.FromArgb(r.Next(255),r.Next(255),r.Next(255));
10:    Color cB=Color.FromArgb(r.Next(255),r.Next(255),r.Next(255));
11:    Point pA=new Point(r.Next(this.ClientSize.Width),
12:                         r.Next(this.ClientSize.Height));
13:    Point pB=new Point(r.Next(this.ClientSize.Width),
14:                         r.Next(this.ClientSize.Height));
15:    LinearGradientBrush brush = new LinearGradientBrush(pA,pB,cA,cB);
16:    Pen p=new Pen(brush,(float)r.NextDouble()*10);
17:    e.Graphics.DrawLine(p,pA,pB);
18:    p.Dispose();
19: }
20: 
21: protected override void OnPaintBackground(PaintEventArgs e)
22: {
23:     // When we resize or on the first time run
24:     // we'll paint the background, otherwise
25:     // it will be left so the lines build up
26:     if(BackgroundDirty)
27:     {
28:         BackgroundDirty = false;
29:         LinearGradientBrush gb=
30:             new LinearGradientBrush(this.ClientRectangle,
31:                                     Color.Navy,
32:                                     Color.Aquamarine,
33:                                     90);
34:         e.Graphics.FillRectangle(gb,this.ClientRectangle);
35:         gb.Dispose();
36:     }
37: }
38: }
```

3.5

GDI+ THE .NET
GRAPHICS
INTERFACE

Figure 3.5.1 shows the application running in all its garish glory.

**FIGURE 3.5.1**

The modified *DrawLines* program.

NOTE

You might notice that the pen itself uses a brush for filling its internal color. You can also use a hatch brush or a pattern brush to draw a line. This is very powerful indeed.

Textured Pens and Brushes

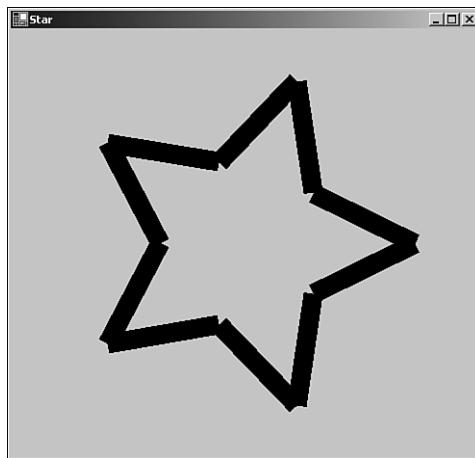
You saw in the previous example that a pen can be given a brush with which to paint the line. Brushes can be solid, gradients, textures, or hatch patterns. A textured brush can be used to draw lines that are painted with a bitmap image. To accomplish this, you can load in a bitmap from a file or a resource, create a brush with it, pass the brush to a pen, and finally draw your lines with the pen. The following snippet of code shows this process:

```
Image i=Image.FromFile("stone.bmp");
TextureBrush b= new TextureBrush(i);
Pen p=new Pen(b,10);
e.Graphics.DrawLine(p,0,0,100,100);
p.Dispose();
```

As you can see, a ridiculously simple thing to do.

Tidying up Your Lines with Endcaps

The line segments shown in Figure 3.5.1 have ends that are cut squarely and might not be good for drawing polygons. Figure 3.5.2 shows how wide lines with square ends look when used to draw a simple star.



3.5

GDI+: THE .NET
GRAPHICS
INTERFACE

FIGURE 3.5.2

Flat end caps used in a polygon.

As you can see, the flat end caps make a mess of the shape where the lines join. To tidy this up, we can specify a rounded end cap for the lines. Pens have two properties, `StartCap` and `EndCap`, that can be set to draw specialized ends on your lines. In Listing 3.5.3, we show the program that draws the star and allows you to turn end caps on or off with a button on the form.

LISTING 3.5.3 DrawStar.cs: The Line Cap Example

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Windows.Forms;
5:
6: class drawstar : System.Windows.Forms.Form
7: {
8:     Button button1;
9:
10:    bool EndCap;
```

LISTING 3.5.3 Continued

```
11:
12:    public void OnPaint(object sender, PaintEventArgs e)
13:    {
14:        Pen pen=new Pen(Color.Black,(float)20.0);
15:        if(EndCap)
16:        {
17:            pen.StartCap=LineCap.Round;
18:            pen.EndCap=LineCap.Round;
19:        }
20:
21:        float r36 = (float)(36.0 * 3.1415926 / 180);
22:        Point Center=new Point(this.ClientRectangle.Width/2,
23:                               this.ClientRectangle.Height/2);
24:        float Pointsize = (float)0.8*Math.Min(Center.X,Center.Y);
25:        for(int i=0; i<10; i++)
26:        {
27:            float d1=(i & 1)==1 ? Pointsize / 2 : Pointsize;
28:            float d2=(i & 1)==0 ? Pointsize / 2 : Pointsize;
29:            e.Graphics.DrawLine(pen,
30:                               new Point((int)(d1*Math.Cos(r36*i))+Center.X,
31:                                         (int)(d1*Math.Sin(r36*i))+Center.Y),
32:                               new Point((int)(d2*Math.Cos(r36*(i+1)))+Center.X,
33:                                         (int)(d2*Math.Sin(r36*(i+1))+Center.Y)));
34:        }
35:        pen.Dispose();
36:    }
37:
38:    public void OnSizeChanged(object sender, EventArgs e)
39:    {
40:        Invalidate();
41:    }
42:
43:    public void OnClickedButton1(object sender, EventArgs e)
44:    {
45:        EndCap = !EndCap;
46:        Invalidate();
47:    }
48:
49:    public drawstar()
50:    {
51:        this.Paint+=new PaintEventHandler(OnPaint);
52:        this.Text="Star";
53:        this.SizeChanged+=new EventHandler(OnSizeChanged);
54:
```

LISTING 3.5.3 Continued

```
55:     button1 = new Button();
56:     button1.Location=new Point(5,5);
57:     button1.Size=new Size(50,20);
58:     button1.Text = "Caps";
59:     button1.Click+=new EventHandler(OnClickedButton1);
60:     this.Controls.Add(button1);
61:
62: }
63:
64: static void Main()
65: {
66:     Application.Run(new drawstar());
67: }
68: };
```

Compile this program with the following command line:

```
csc /t:winexe drawstar.cs
```

When you run the program, you will see a button in the top-left corner of the window, click it to change to rounded end caps. There are 11 different line cap styles altogether, including a custom style.

Already you can see that GDI+ is a great improvement over good old trusty GDI, and we've only really looked at lines and rectangles. Let's now look at curves, paths, and splines.

Curves and Paths

GDI has had Bezier curves and temporary paths for some time. GDI+ expands these capabilities by providing a rich set of functionality for curves and a persistent path object.

The simplest form is the curve drawn from an array of points. The curve is drawn so that it passes through each point on the array. The tension of the curve for any particular point is controlled by the positions of the points on either side of it in the array. The true Bezier curve has two points associated with each node—a position and a control point. Tension of the curve is determined by the relationship of the control point to the direction of the line.

Listing 3.5.4 illustrates the use of the `Graphics.DrawCurve` method by creating a set of positions that bounce around in a rectangular area onscreen. A curve, or Cardinal Spline, is drawn along this array of points, and you can select the tension of the curve with a slider.

LISTING 3.5.4 DrawCurve.cs: Using the DrawCurve Method

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace Curves
10: {
11:     /// <summary>
12:     /// This simple object bounces points
13:     /// around a rectangular area.
14:     /// </summary>
15:     class Bouncer
16:     {
17:         int dirx;
18:         int diry;
19:         int X;
20:         int Y;
21:
22:         Size size;
23:
24:         public Bouncer()
25:         {
26:             dirx=diry=1;
27:         }
28:
29:         public void Move()
30:         {
31:             X+=dirx;
32:             Y+=diry;
33:
34:             if(X<=0 || X>=size.Width)
35:                 dirx*=-1;
36:
37:             if(Y<=0 || Y>=size.Height)
38:                 diry*=-1;
39:         }
40:
41:         public Point Position
42:         {
43:             get{return new Point(X,Y);}
44:             set{X=value.X; Y=value.Y;}
45:         }
}
```

LISTING 3.5.4 Continued

```
46:
47:     public Size Size
48:     {
49:         get{ return size; }
50:         set{size = value; }
51:     }
52: }
53:
54: /// <summary>
55: /// Summary description for Form1.
56: /// </summary>
57: public class Curves : System.Windows.Forms.Form
58: {
59:
60:     Timer bounce,paint;
61:     TrackBar trk;
62:
63:     Bouncer[] bouncers;
64:
65:     void OnTickBounce(object sender, EventArgs e)
66:     {
67:         foreach(Bouncer b in bouncers)
68:             b.Move();
69:     }
70:
71:     void OnTickPaint(object sender, EventArgs e)
72:     {
73:         Invalidate();
74:     }
75:
76:     public void OnPaint(object sender, PaintEventArgs e)
77:     {
78:         Pen p=new Pen(Color.Red,10);
79:         SolidBrush br=new SolidBrush(Color.Blue);
80:         Point[] points = new Point[bouncers.Length];
81:         int i=0;
82:         // need to translate our bouncers to an array of points
83:         foreach(Bouncer b in bouncers)
84:             points[i++]=b.Position;
85:         //Draw the curve
86:         e.Graphics.DrawCurve(p,points,0,points.Length-
➥1,(float)trk.Value);
87:         //now draw the nodes in the curve.
88:         foreach(Point pn in points)
89:             e.Graphics.FillEllipse(br,pn.X-5,pn.Y-5,10,10);
```

LISTING 3.5.4 Continued

```
90:         p.Dispose();
91:         br.Dispose();
92:     }
93:
94:     public Curves()
95:     {
96:         this.Paint+=new PaintEventHandler(OnPaint);
97:         // A timer to manage the bouncing
98:         bounce=new Timer();
99:         bounce.Interval=5;
100:        bounce.Tick+=new EventHandler(OnTickBounce);
101:        // A timer to manage the painting refresh
102:        paint=new Timer();
103:        paint.Interval=100;
104:        paint.Tick+=new EventHandler(OnTickPaint);
105:        // Random number generator for initial positions
106:        Random r=new Random();
107:        // the form initial size
108:        this.Size=new Size(800,600);
109:        //initialize an array of bouncing points
110:        bouncers = new Bouncer[6];
111:        for(int i=0;i<6;i++)
112:        {
113:            bouncers[i]=new Bouncer();
114:            bouncers[i].Position=new Point(r.Next(800),r.Next(600));
115:            bouncers[i].Size=new Size(800,600);
116:        }
117:        // turn on the timers
118:        bounce.Enabled=true;
119:        paint.Enabled=true;
120:        //create a trackbar for the line tension
121:        trk = new TrackBar();
122:        trk.Location=new Point(5,25);
123:        trk.Size=new Size(100,20);
124:        trk.Minimum=1;
125:        trk.Maximum=10;
126:        trk.Value=2;
127:        this.Controls.Add(trk);
128:        //and label it nicely for the user
129:        Label lb=new Label();
130:        lb.Location=new Point(5,5);
131:        lb.Size=new Size(100,20);
132:        lb.Text="Curve tension";
133:        this.Controls.Add(lb);
134:    }
```

LISTING 3.5.4 Continued

```
135:  
136:     static void Main()  
137:     {  
138:         Application.Run(new Curves());  
139:     }  
140: }  
141: }
```

Compile Listing 3.5.4 with the following command line:

```
Csc /t:winexe drawcurve.cs
```

The work is done by two timers and a paint handler. Timer #1 (lines 98 and 65–69) makes the positions of the curve nodes bounce around the screen. Timer #2 (lines 102 and 71–74) invalidates the screen so that the painting is not done too frequently. The paint handler (lines 76–90) creates an array of points from the bounce object; sadly, `Point` is a sealed class so you cannot inherit from it, which would have been useful. Then line 86 draws the curve, the tension of which depends on the position of the `TrackBar` control. To illustrate where the individual points are, lines 88 and 89 draw them in blue. This demonstration clearly shows how line tension works for the simple curve.

Figure 3.5.3 shows the `DrawCurves` application running.

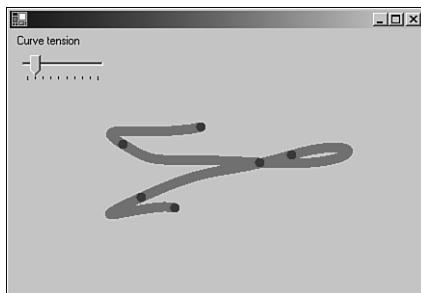


FIGURE 3.5.3

Drawing a curve with tension.

A Bezier Spline is somewhat different. For each line segment, two points define the start and end of the line, and another two determine the control points that add the line tension and curvature. Listing 3.5.5 shows how a Bezier curve is used.

LISTING 3.5.5 Beziercurves.cs: Using the Bezier Curve

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace Curves
10: {
11:     /// <summary>
12:     /// This simple object bounces points
13:     /// around a rectangular area.
14:     /// </summary>
15:     class Bouncer
16:     {
17:         int dirx;
18:         int diry;
19:         public int X;
20:         public int Y;
21:
22:         Size size;
23:
24:         public Bouncer()
25:         {
26:             dirx=diry=1;
27:         }
28:
29:         public void Move()
30:         {
31:             X+=dirx;
32:             Y+=diry;
33:
34:             if(X<=0 || X>=size.Width)
35:                 dirx*=-1;
36:
37:             if(Y<=0 || Y>=size.Height)
38:                 diry*=-1;
39:         }
40:
41:         public Point Position
42:         {
43:             get{return new Point(X,Y);}
44:             set{X=value.X; Y=value.Y;}
45:         }
}
```

LISTING 3.5.5 Continued

```
46:  
47:     public Size Size  
48:     {  
49:         get{ return size;}  
50:         set{size = value;}  
51:     }  
52: }  
53:  
54: /// <summary>  
55: /// Summary description for Form1.  
56: /// </summary>  
57: public class BezierCurves : System.Windows.Forms.Form  
58: {  
59:  
60:     Timer bounce,paint;  
61:     Bouncer[] bouncers;  
62:  
63:     void OnTickBounce(object sender, EventArgs e)  
64:     {  
65:         foreach(Bouncer b in bouncers)  
66:             b.Move();  
67:     }  
68:  
69:     void OnTickPaint(object sender, EventArgs e)  
70:     {  
71:         Invalidate();  
72:     }  
73:  
74:     public void OnPaint(object sender, PaintEventArgs e)  
75:     {  
76:         Pen p=new Pen(Color.Red,10);  
77:         p.StartCap=LineCap.DiamondAnchor;  
78:         p.EndCap=LineCap.ArrowAnchor;  
79:         SolidBrush br=new SolidBrush(Color.Blue);  
80:         //Draw the curve  
81:         e.Graphics.DrawBezier(p,new Point(550,300),  
82:                               bouncers[0].Position,  
83:                               bouncers[1].Position,  
84:                               new Point(50,300));  
85:         //now draw the nodes in the curve.  
86:         foreach(Bouncer b in bouncers)  
87:             e.Graphics.FillEllipse(br,b.X-5,b.Y-5,10,10);  
88:         //and show the relation between the bouncing node  
89:         //and the bezier end point
```

LISTING 3.5.5 Continued

```
90:         p=new Pen(Color.Black,1);
91:         p.DashStyle=DashStyle.DashDotDot;
92:         e.Graphics.DrawLine(p,bouncers[0].Position,new Point(550,300));
93:         e.Graphics.DrawLine(p,bouncers[1].Position,new Point(50,300));
94:         p.Dispose();
95:     }
96:
97:     public BezierCurves()
98:     {
99:         this.Paint+=new PaintEventHandler(OnPaint);
100:        // A timer to manage the bouncing
101:        bounce=new Timer();
102:        bounce.Interval=5;
103:        bounce.Tick+=new EventHandler(OnTickBounce);
104:        // A timer to manage the painting refresh
105:        paint=new Timer();
106:        paint.Interval=100;
107:        paint.Tick+=new EventHandler(OnTickPaint);
108:        // Random number generator for initial positions
109:        Random r=new Random();
110:        // the form initial size
111:        this.Size=new Size(800,600);
112:        //initialize an array of bouncing points
113:        bouncers = new Bouncer[2];
114:        for(int i=0;i<2;i++)
115:        {
116:            bouncers[i]=new Bouncer();
117:            bouncers[i].Position=new Point(r.Next(800),r.Next(600));
118:            bouncers[i].Size=new Size(800,600);
119:        }
120:        // turn on the timers
121:        bounce.Enabled=true;
122:        paint.Enabled=true;
123:    }
124:
125:    static void Main()
126:    {
127:        Application.Run(new BezierCurves());
128:    }
129: }
130: }
```

The meat of this application is substantially similar to that shown in Listing 3.5.4. The points of interest are lines 83–86, where the Bezier is drawn, and lines 92–96, which show you how to draw dashed lines. The image shown in Figure 3.5.4 is a screenshot of the application and shows how the Bezier control points are used to add direction and tension to a particular node.

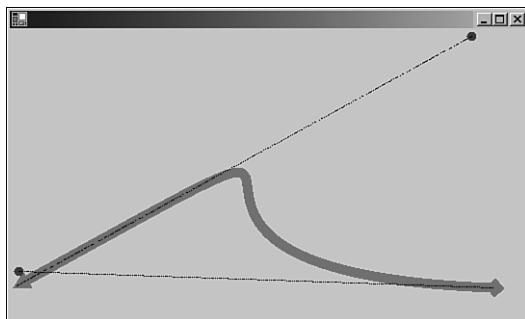


FIGURE 3.5.4
The Bezier curve in action.

3.5

GDI+ THE .NET GRAPHICS INTERFACE

An array of points can also be used to create a multi-segmented Bezier Spline. The array must have a multiple of four points and is arranged as follows:

- Point[0] = Start point of line segment 1
 - Point[1] = Control point for the start point
 - Point[2] = Control point for the end point
 - Point[3] = End point for line segment and start point of line segment 2
 - Point[4] = Control point for the start point of line segment 2
- And so on, giving 4 initial points plus three for each subsequent line segment.

The GraphicsPath Object

GDI+ Graphics Paths are a convenient way of collecting together a number of graphical shapes, or their boundaries at least, into a single unit. A path, once created, can be manipulated in its entirety, filled, stroked, or used to perform other graphical operations, such as being used to create a clipping region.

Any combination of the following graphical primitives can be placed in a Path object:

- Arcs
- Bezier Splines
- Cardinal splines

- Ellipses
- Lines
- Paths
- Pie segments
- Polygons
- Rectangles
- Character Glyphs from Strings

The following code snippet produces the result seen in Figure 3.5.5.

```
void OnPaint(object sender, PaintEventArgs e)
{
    GraphicsPath p=new GraphicsPath();
    p.AddEllipse(10,10,100,100);
    p.AddRectangle(new Rectangle(0,0,120,120));
    Pen pen=new Pen(Color.Black,1);
    e.Graphics.DrawPath(pen,p);
    pen.Dispose();
}
```

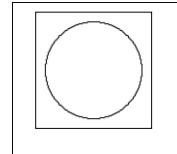


FIGURE 3.5.5

Two figures in a GraphicsPath.

Filling the path by substituting a suitable brush and fill command produces the following effect (see Figure 3.5.6):

```
SolidBrush brush = new SolidBrush(Color.Blue);
e.Graphics.FillPath(brush,p);
```

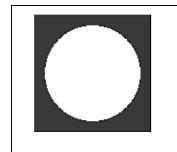


FIGURE 3.5.6

A filled path.

The square part of the path has been filled and the circle has not. This is because the graphics `FillMode` for the path is set to the default setting, `Alternate`.

To fill everything inside the outermost boundary of the shape, set the `FillMode` to `Winding`.

Adding Text and Other Paths

One path can be added to another quite simply by calling the `AddPath` method. The following code snippet creates a second path and adds it to the one shown in Figure 3.5.5.

```
void OnPaint(object sender, PaintEventArgs e)
{
    GraphicsPath p=new GraphicsPath();
    p.AddEllipse(10,10,100,100);
    p.AddRectangle(new Rectangle(0,0,120,120));
    Pen pen=new Pen(Color.Black,1);
    GraphicsPath p2=new GraphicsPath();
    Point[] tripoint=new Point[3];
    tripoint[0]=new Point(80,10);
    tripoint[1]=new Point(80,110);
    tripoint[2]=new Point(150,60);
    p2.AddClosedCurve(tripoint,(float)0);
    p2.AddPath(p,true);
    SolidBrush brush = new SolidBrush(Color.Blue);
    e.Graphics.FillPath(brush,p2);
    e.Graphics.DrawPath(pen,p2);
}
```

Now the image displayed is the sum of the two paths. Figure 3.5.7 shows the output from the modified `OnPaint` handler.



FIGURE 3.5.7
The combination of two paths

Placing text in a path is a great way of creating text effects. A text path only contains the glyph outlines of the letters used. These outlines can be used to create clip paths, filled with patterns or colors, scaled, rotated, or otherwise transformed to make some pretty impressive effects.

Modifying the code in our phantom `OnPaint` will illustrate again how this is accomplished.

```
void OnPaint(object sender, PaintEventArgs e)
{
    GraphicsPath p=new GraphicsPath();
    p.AddString("AYBABTU",FontFamily.GenericSansSerif,
                0,(float)72,new Point(0,0),
                StringFormat.GenericDefault);
    SolidBrush brush = new SolidBrush(Color.Blue);
    e.Graphics.FillPath(brush,p);
    brush.Dispose();
}
```

We'll continue this discussion in the next section because it leads nicely into that topic.

Clipping with Paths and Regions

A region is a description of some enclosed shape that can be used as a mask in which to perform graphical operations. Regions can be regular shapes, like rectangles or ellipses; they can also be irregular, perhaps created from curves or text glyphs. Because regions can be created from paths, it is possible to have very complex, clipped shapes. Going back again to our previous text path example, we can create a clipping region from it and use it for other interesting things.

In the example shown in Listing 3.5.6, we use a path, filled with a text string, as a mask for randomly positioned ellipses of color that splatter the window but are clipped by the glyph outlines of the text.

LISTING 3.5.6 ClipToPath.cs: Using a Path to Clip the Drawing Area

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace Clipping
10: {
11:     public class ClipToPath : System.Windows.Forms.Form
12:     {
13:
14:         Timer timer;
15:         GraphicsPath p;
16:         bool dirty;
```

LISTING 3.5.6 Continued

```
17:         void OnPaint(object sender, PaintEventArgs e)
18:         {
19:             Random r=new Random();
20:             SolidBrush brush = new SolidBrush(
21:                 Color.FromArgb(r.Next(255),
22:                                 r.Next(255),
23:                                 r.Next(255)));
24:             e.Graphics.SetClip(p);
25:             e.Graphics.FillEllipse(brush,
26:                                 r.Next(500),
27:                                 r.Next(200),
28:                                 r.Next(20),
29:                                 r.Next(20));
30:             brush.Dispose();
31:         }
32:     }
33:
34:     void OnTick(object sender, EventArgs e)
35:     {
36:         Invalidate();
37:     }
38:
39:     protected override void OnPaintBackground(PaintEventArgs e)
40:     {
41:         if(dirty)
42:         {
43:             e.Graphics.ResetClip();
44:             SolidBrush b=new SolidBrush(this.BackColor);
45:             e.Graphics.FillRectangle(b, this.ClientRectangle);
46:             dirty = false;
47:             b.Dispose();
48:         }
49:     }
50:
51:     void OnSized(object sender, EventArgs e)
52:     {
53:         dirty=true;
54:         Invalidate();
55:     }
56:
57:     public ClipToPath()
58:     {
59:
60:         p=new GraphicsPath();
61:         p.AddString("AYBABTU",FontFamily.GenericSansSerif,
```

LISTING 3.5.6 Continued

```
62:                               0,(float)72,new Point(0,0),
63:                               StringFormat.GenericDefault);
64:               dirty=true;
65:               this.Paint+=new PaintEventHandler(OnPaint);
66:               this.SizeChanged+=new EventHandler(OnSized);
67:               timer = new Timer();
68:               timer.Interval=10;
69:               timer.Tick+=new EventHandler(OnTick);
70:               timer.Enabled=true;
71:           }
72:
73:           static void Main()
74:           {
75:               Application.Run(new ClipToPath());
76:           }
77:       }
78:   }
```

The path is created once on lines 59–62 and stored for use later. Then a timer is initialized to enable the periodic refresh of the page.

Lines 39–48 handle the repaint of the background if the window is resized or on the first draw.

Finally, line 25 selects the clip-path into the Graphics object. The following lines place a random blob of color on the page; this might or might not be seen, depending on its coincidence with the clipping path.

After a while, the output from the application looks something like the image in Figure 3.5.8.



FIGURE 3.5.8

Painting clipped to a path.

Operations on regions are different from those on paths. A path specifies a set of boundaries, a region specifies an area or group of areas to be used as a mask. Regions can be combined in several ways. The operation is controlled by the `CombineMode` enumeration. Listing 3.5.7 illustrates the use of regions in four different modes.

LISTING 3.5.7 Regions.cs: Combining Regions in Different Ways

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace regions
10: {
11:     /// <summary>
12:     /// Summary description for Form1.
13:     /// </summary>
14:     public class RegionsTest : System.Windows.Forms.Form
15:     {
16:         void PaintRegions(Graphics g,CombineMode m,
17:             Point offset,string text)
18:         {
19:             Region ra,rb;
20:             GraphicsPath p=new GraphicsPath();
21:             p.AddRectangle(new Rectangle(60,60,120,120));
22:             ra=new Region(p);
23:             p=new GraphicsPath();
24:             p.AddEllipse(0,0,120,120);
25:             rb=new Region(p);
26:             ra.Translate(offset.X,offset.Y );
27:             rb.Translate(offset.X,offset.Y);
28:             g.SetClip(ra,CombineMode.Replace);
29:             g.SetClip(rb,m);
30:             SolidBrush brush=new SolidBrush(Color.Black);
31:             g.FillRectangle(brush,this.ClientRectangle);
32:             g.ResetClip();
33:             g.DrawString(text,
34:                 new Font("Arial",(float)16),
35:                 brush,(float)offset.X+60,(float)offset.Y+220);
36:         }
37:         void OnPaint(object sender, PaintEventArgs e)
38:         {
39:             // A union of two regions...
40:             PaintRegions(e.Graphics,CombineMode.Union,
41:                 new Point(0,0),"Union");
42:             // the intersection of two regions
43:             PaintRegions(e.Graphics,CombineMode.Intersect,
44:                 new Point(250,0),"Intersect");
```

LISTING 3.5.7 Continued

```
45:           // Region exclusive or
46:           PaintRegions(e.Graphics,CombineMode.Xor,
47:                           new Point(500,0),"Xor");
48:           // The complement of the two regions
49:           PaintRegions(e.Graphics,CombineMode.Complement,
50:                           new Point(0,250),"Complement");
51:           // Exclusion.
52:           PaintRegions(e.Graphics,CombineMode.Exclude,
53:                           new Point(250,250),"Exclude");
54:       }
55:
56:       public RegionsTest()
57:       {
58:           this.Paint+=new PaintEventHandler(OnPaint);
59:           this.Size = new Size(800,800);
60:       }
61:
62:       static void Main()
63:       {
64:           Application.Run(new RegionsTest());
65:       }
66:   }
67: }
```

OnPaint (lines 37–54) repeatedly calls the PaintRegions method with a distinct CombineMode setting, some positional information, and an identifying string. The PaintRegions method (lines 16–35) begins by declaring two regions, *ra* and *rb*, and filling them with a circle and a rectangle. Lines 25 and 26 ensure that the regions are moved into position ready for display, and then line 27 selects the first clip region, replacing the one that already exists in the Graphics object. The second region is combined with the CombineMode, supplied on line 28, and the whole rectangle of the client area filled with black. Note that only the area inside the clipped region is filled. The program then removes the clipping region on line 31 to draw the information text on lines 32–34.

Output from this program is shown in Figure 3.5.9.

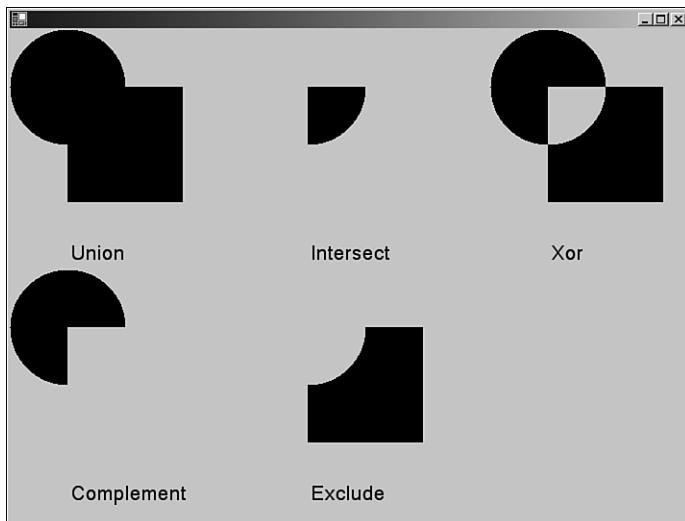


FIGURE 3.5.9
Exploring region combination modes.

3.5

GDI+: THE .NET
GRAPHICS
INTERFACE

Transformations

You might have noticed a couple of commands in the previous example that moved the regions into position by applying a translation to them. There are two basic ways of applying transforms to graphical objects in the Framework. You can use the methods provided, such as `Translate` or `Rotate`, or you can specify an explicit transformation matrix to be used.

By far, the easiest way for dealing with these kinds of operations is to use the methods that wrap the underlying matrix manipulations for you.

These operations allow you to do the following:

- **Translate**—Move an object in the x or y plane by an offset.
- **Rotate**—Spin an object about the origin.
- **RotateAt**—Spin an object about a point other than the origin.
- **Scale**—Magnify or reduce an object in the x and y planes.
- **Shear**—Distort an object by changing its bounding rectangle into a bounding parallelogram.

A matrix performs these operations by performing a matrix operation, such as addition, subtraction, or multiplication to the current graphics transform. We mentioned right at the beginning of this chapter that graphical commands need to go through several translations

between being drawn into the graphics system and finally being seen onscreen. This is referred to as the *graphics pipeline*. Think of commands going in one end, being changed according to the plumbing of the pipes, and emerging at the other end in a somewhat different state.

The things a matrix transforms are the individual positional coordinates of a graphical object. Every pixel drawn onscreen will have been transformed in the pipeline several times before it finally appears on the screen or printer.

The matrices used in the framework are a two dimensional 3×3 matrix. The matrix is constructed as shown in Figure 3.5.10.

The 3×3 matrix used by the >NET Matrix class

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Linear part
Translation part
Always 0,0,1

FIGURE 3.5.10

The 3×3 Identity Matrix.

The matrix shown in Figure 3.5.10 is called the Identity Matrix. An identity matrix can be applied as a transformation to an object and have no effect on it. The Identity Matrix is normally the starting point for all transformations. The matrix used by GDI+ has some fixed values in the rightmost column. The part that is always 0,0,1 in the 3×3 matrix is used to allow a compound operation (for example, a linear calculation, such as rotation or scaling) to be followed by a translation in the same multiplication. This is known as an *affined matrix calculation*. To do an affined calculation on a matrix of n dimensions, the multiplication matrix used must be $n+1$ by $n+1$. Therefore, a 2D calculation requires a 3×3 matrix. For this reason, the third column of the matrix is always set to 0,0,1, and you cannot change it.

There are lots of books on graphics but, for the sake of completeness, the operations performed by the matrix on coordinates go as follows. The following example does a simple translation by an X and Y amount.

A coordinate is turned into a vector by adding a third, z column, to it. For example,

[10,5] becomes [10,5,1]

The dx and dy, or the translation part, of the matrix shown is 10,30.

This vector is multiplied by the matrix by working down each column in the following manner.

[10 , 5 , 1]

multiply...

1 0 0

0 1 0

dX dY 1

equals...

1*x 0*x 0*x

+++

0*y 1*y 0*y

+++

1*dX 1*dY 1*1

equals...

dx+x dy+y 1

equals...

20 35 1

Take the dummy component out, and you're left with [20,35], which is [10,5] translated by [10,30].

A rotation about the origin is also performed in a similar manner. We'll use 90 degrees, because the sine and cosine of 90 are easy to compute. The matrix for rotation is initialized as follows:

$\cos\Theta \sin\Theta \ 0$

$-\sin\Theta \cos\Theta \ 0$

0 0 1

So, $\cos(90)=0$ and $\sin(90)=1$ so, remembering to work down the columns, the matrix calculation looks like the following:

[10 , 5 , 1]

multiply...

0 1 0

-1 0 0

0 0 1

equals...

0*x 1*x 0*x

+++

-1*y 0*y 0*y

+++

1*0 1*0 1*1

equals...

[-5 , 10 , 1]

Chop off the extraneous 1, and you get [-5,10]

Taking two matrices and adding them or multiplying them together produces a resultant matrix. Successive additions or multiplications create a matrix that is the sum of all the operations performed so far. This means that you can do several operations on a matrix and they all accumulate, and then the transform in the matrix is applied to each and every pixel that the drawing command produces to obtain their resulting positions in the final output.

The order in which operations take place is very important too. For example, `Rotate—Scale—Translate` does not mean the same thing as `Scale—Translate—Rotate`. This implies that you also need to think about how the API commands apply the matrices you hand them. Do you multiply the current matrix by the one you just supplied or the one you have by the current matrix? Luckily, the calls to `Rotate`, `Scale`, and so on, have a flag that you can use to control the way matrices are worked on. `MatrixOrder.Prepend`, the default, applies the matrix you pass first, and then the current matrix. `MatrixOrder.Append` applies the requested operation after the current matrix is applied.

The following sequence illustrates the contents of a matrix as it evolves through many operations.

```
Matrix m=new Matrix() // Create an identity matrix
```

1 0 0

0 1 0

0 0 1

```
m.Rotate(30,MatrixOrder.Append); //rotate 30 degrees  
0.8660254 0.5 0  
-0.5 0.8660254 0  
0 0 1  
  
m.Scale((float)1,(float)3,MatrixOrder.Append); //magnify by 3 in the Y plane  
0.8660254 1.5 0  
-0.5 2.598076 0  
0 0 1  
  
m.Translate((float)100,(float)130,MatrixOrder.Append); //move by 100 X and 130  
Y  
0.8660254 1.5 0  
-0.5 2.598076 0  
100 130 1
```

3.5

As this sequence progresses, you can see how the matrix accumulates the operations with each successive call.

It is important to note that keeping track of the graphics matrix is very important, especially if you want to place many different objects onscreen, each with its own transformation. It is sometimes useful to put the current matrix into a known state, perhaps with the origin in a different place or zoomed in or out by scaling, and then perform other operations. Each operation should behave itself and leave the current matrix as it was found. Saving the state of the `Graphics` object in a `GraphicsState` can do this. A `GraphicsState` is filled with information by the `Graphics.Save()` method and restored with the `Graphics.Restore(state)` method. For example,

```
GraphicsState gs=theGraphics.Save();  
// perform operations here...  
theGraphics.Restore(gs);  
//Graphics are back to their original state.
```

Listing 3.5.8 demonstrates the simple transformation sequence discussed previously, as well as the use of `GraphicsState` and other matrix manipulations in the context of some simple graphic shapes.

LISTING 3.5.8 MatrixElements.cs: A Scene from the Matrix

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Drawing.Text;
5: using System.Collections;
6: using System.ComponentModel;
7: using System.Windows.Forms;
8: using System.Data;
9: using System.Text;
10:
11: namespace matrixelements
12: {
13:     public class MatrixElements : System.Windows.Forms.Form
14:     {
15:         void DumpMatrix(Graphics g, Matrix m, Point p)
16:         {
17:             Stringbuilder sb=new Stringbuilder();
18:             sb.AppendFormat("{0},\t{1},\t{2},\t{3},\t{4},\t{5},\t{6}",
19:                 m.Elements[0],m.Elements[1],m.Elements[2],
20:                 m.Elements[3],m.Elements[4],m.Elements[5]);
21:             GraphicsState s=g.Save();
22:             g.ResetTransform();
23:             g.DrawString(sb.ToString(),new Font("Courier New", (float)16),
24:                         new SolidBrush(Color.Black),p,StringFormat.GenericDefault);
25:             g.Restore(s);
26:         }
27:
28:         void OnSize(object sender,EventArgs e)
29:         {
30:             Invalidate();
31:         }
32:
33:         void OnPaint(object sender, PaintEventArgs e)
34:         {
35:             GraphicsState gs;
36:             Matrix m=new Matrix();
37:
38:             //position and draw the axes by translating the whole window
39:             // so that the origin is in the center of the screen
40:             e.Graphics.TranslateTransform((float)this.ClientRectangle.Width/2,
41:                                         (float)this.ClientRectangle.Height/2);
42:             e.Graphics.DrawLine(new Pen(Color.Black,(float)1),0,-1000,0,1000);
43:             e.Graphics.DrawLine(new Pen(Color.Black,(float)1),-100,0,1000,0);
44:
45:             //Draw an ordinary square about the origin.
46:             e.Graphics.DrawRectangle(new Pen(Color.Black,(float)3),
47:                                     -50,-50,100,100);
```

LISTING 3.5.8 Continued

```
48:     DumpMatrix(e.Graphics,m,new Point(0,0));
49:
50:     m.Rotate(30,MatrixOrder.Append);
51:     DumpMatrix(e.Graphics,m,new Point(0,100));
52:
53:     gs=e.Graphics.Save();
54:     e.Graphics.MultiplyTransform(m);
55:     e.Graphics.DrawRectangle(new Pen(Color.Red,3),
56:                             -50,-50,100,100);
57:     e.Graphics.Restore(gs);
58:
59:     m.Scale(1,3,MatrixOrder.Append);
60:     DumpMatrix(e.Graphics,m,new Point(0,200));
61:
62:     gs=e.Graphics.Save();
63:     e.Graphics.MultiplyTransform(m);
64:     e.Graphics.DrawRectangle(new Pen(Color.Green,3),
65:                             -50,-50,100,100);
66:     e.Graphics.Restore(gs);
67:
68:     m.Translate(100,130,MatrixOrder.Append);
69:     DumpMatrix(e.Graphics,m,new Point(0,300));
70:
71:     gs=e.Graphics.Save();
72:     e.Graphics.MultiplyTransform(m);
73:     e.Graphics.DrawRectangle(new Pen(Color.Blue,3),
74:                             -50,-50,100,100);
75:     e.Graphics.Restore(gs);
76: }
77:
78: public MatrixElements()
79: {
80:     this.Paint+=new PaintEventHandler(OnPaint);
81:     this.SizeChanged+=new EventHandler(OnSize);
82: }
83:
84: static void Main()
85: {
86:     Application.Run(new MatrixElements());
87: }
88: }
89: }
```

Compile the code using the following command line:

```
csc /t:winexe matrixelements.cs
```

Cutting to the chase, lines 40 and 41 create a matrix that shifts the origin to the center of the window. Lines 42 and 43 draw axis lines just to look nice and give a point of reference.

Lines 46 and 47 draw our reference square. This is the same drawing command used for all the other objects onscreen, and then line 48 dumps our identity matrix.

Line 50 rotates the matrix, which is again dumped to text, and then line 53 saves the graphics state so as not to mess up the origin. Line 54 uses the matrix, lines 55 and 56 draw our newly rotated square, and line 57 restores the world transform back to its origin-in-the-middle state.

Line 59 scales the matrix, making it 3 times taller than it is wide and stretching the output in the Y direction. Lines 62–66 use the transformation, draw the square, and return the world transform to our chosen default.

Line 68 then translates the matrix moving to the right and down. Line 69 shows the matrix settings and we round-off on lines 71–75 drawing the last, rotated, scaled, and translated square.

To draw the matrix data itself, lines 15–26 save the graphics state and reset the transform, place the text onscreen, and then returns the transform back to its original state before returning.

The final output from this program is shown in Figure 3.5.11.

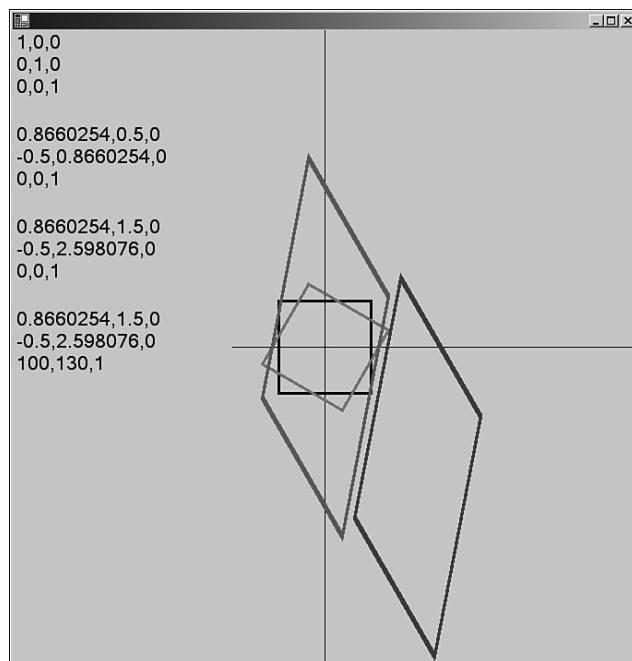


FIGURE 3.5.11

Matrix transformations of a simple square.

Alpha Blending

All colors in GDI+ have a fourth component in addition to the normal red, green, and blue values. This value is the “alpha” and it controls the amount that the background shows through the object just placed onscreen.

Alpha blending can be applied to all graphical shapes, such as lines and text. It can also be applied to images, either as a global value that affects the appearance of the whole image, or as a value for each individual pixel, which makes parts of the image more transparent than others.

To create a color with an alpha component, you can use the `Color.FromArgb(...)` method and supply the alpha as a value from 0, totally transparent, to 255, which is opaque.

Listing 3.5.9 demonstrates this effect by creating an array of ellipses that let the background show through according to their alpha value.

LISTING 3.5.9 AlphaBlend.cs: Using a Solid Brush with Alpha

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace Alphablending
10: {
11:     public class Alphablending : System.Windows.Forms.Form
12:     {
13:
14:         void OnPaint(object sender, PaintEventArgs e)
15:         {
16:             SolidBrush b=new SolidBrush(Color.Red);
17:             Rectangle r=this.ClientRectangle;
18:             GraphicsPath pth=new GraphicsPath();
19:             for(int c=1;c<10;c++)
20:             {
21:                 r.Inflate(-(this.ClientRectangle.Width/20),
22:                           -(this.ClientRectangle.Height/20));
23:                 pth.AddRectangle(r);
24:             }
25:             e.Graphics.FillPath(b,pth);
26:             Random rnd=new Random();
27:             for(int y=0;y<5;y++)
28:             {
```

LISTING 3.5.9 Continued

```
29:         for(int x=0;x<5;x++)
30:         {
31:             b.Color=Color.FromArgb((int)((((5*x)+y)*10.63)),
32:                                     (byte)rnd.Next(255),
33:                                     (byte)rnd.Next(255),
34:                                     (byte)rnd.Next(255));
35:             e.Graphics.FillEllipse(b,this.ClientRectangle.Width/5*x,
36:                                   this.ClientRectangle.Height/5*y,
37:                                   this.ClientRectangle.Width/5,
38:                                   this.ClientRectangle.Height/5);
39:         }
40:     }
41: }
42:
43: void OnSize(object sender, EventArgs e)
44: {
45:     Invalidate();
46: }
47:
48: public Alphablending()
49: {
50:     this.Paint+=new PaintEventHandler(OnPaint);
51:     this.SizeChanged+=new EventHandler(OnSize);
52:     this.BackColor=Color.White;
53: }
54:
55:
56: static void Main()
57: {
58:     Application.Run(new Alphablending());
59: }
60: }
61: }
```

Compile Listing 3.5.9 with the following command line:

```
csc /t:winexe alphablend.cs
```

The output from Listing 3.5.9 is shown in Figure 3.5.12 and clearly shows the background scene, the dark concentric rectangles, being obscured by the graduated alpha blends of the ellipses.

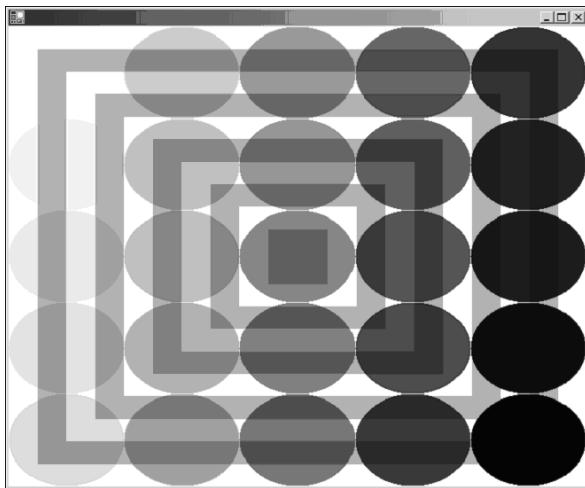


FIGURE 3.5.12
Alpha blending at work.

3.5

GDI+ THE .NET
GRAPHICS
INTERFACE

Alpha Blending of Images

This task is also easy to accomplish using a `ColorMatrix` object. We saw in the section on matrix manipulations that the matrix could be used to perform manipulations in 2D space for graphic objects using a 3×3 matrix. A `ColorMatrix` deals with a four dimensional concept, the color space. A color space's dimensions are R,G,B and A for Red, Green, Blue, and Alpha, respectively. Performing arbitrary matrix manipulations on a four dimensional object requires a 5×5 matrix. Like the 3×3 example for 2D space, the `ColorMatrix` maintains a dummy column that is set to 0,0,0,0,1.

The image in Figure 3.5.13 is the Identity Matrix for the `ColorMatrix` object.

The 5×5 Identity Matrix for the `Color Matrix`

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Linear part
Translation part
Dummy column, always 0,0,0,0,1

FIGURE 3.5.13

The color space Identity Matrix.

The `ColorMatrix` class, like the matrix for 2D manipulations, has properties for its elements that can be accessed individually. To set the alpha for an image, you simply need to put the alpha value into the `Matrix33` property as follows:

```
ColorMatrix cm=new ColorMatrix(); // create an identity matrix  
m.Matrix33=(float)AlphaValue;
```

This color matrix is then handed to an `ImageAttributes` class:

```
ImageAttributes ia=new ImageAttributes();  
ia.SetColorMatrix(m);
```

The initialized `ImageAttributes` object is then used by the `Graphics.DrawImage()` method to paint the image with the specified alpha blend.

Listing 3.5.10 show you how to use the alpha blending features of GDI+ with any bitmap image from your machine. The program has a button to allow you to choose an image file and a track bar that lets you select the amount of transparency with which the image is displayed.

LISTING 3.5.10 ImageAlpha.cs: Alpha Blending an Image

```
1: using System;  
2: using System.Drawing;  
3: using System.Drawing.Drawing2D;  
4: using System.Drawing.Imaging;  
5: using System.Collections;  
6: using System.ComponentModel;  
7: using System.Windows.Forms;  
8: using System.Data;  
9:  
10: namespace ImageAlpha  
11: {  
12:     class Form1 : Form  
13:     {  
14:  
15:         Button b;  
16:         TrackBar t;  
17:         Image i;  
18:  
19:         void OnPaint(object Sender,PaintEventArgs e)  
20:         {  
21:             SolidBrush b=new SolidBrush(Color.Red);  
22:             Rectangle r=this.ClientRectangle;  
23:             GraphicsPath pth=new GraphicsPath();  
24:             for(int c=1;c<10;c++)  
25:             {  
26:                 r.Inflate(-(this.ClientRectangle.Width/20),
```

LISTING 3.5.10 Continued

```
27:             -(this.ClientRectangle.Height/20));
28:             pth.AddRectangle(r);
29:         }
30:         e.Graphics.FillPath(b,pth);
31:
32:         if(i!=null)
33:         {
34:             ColorMatrix m=new ColorMatrix();
35:             m.Matrix33=(float)(1.0/256*t.Value);
36:             ImageAttributes ia=new ImageAttributes();
37:             ia.SetColorMatrix(m);
38:             e.Graphics.DrawImage(i,this.ClientRectangle,
39:             ➔0,0,i.Width,i.Height,GraphicsUnit.Pixel,ia);
40:         }
41:
42:         void OnClickB(object sender, EventArgs e)
43:         {
44:             OpenFileDialog dlg=new OpenFileDialog();
45:             dlg.Filter="Bitmap files (*.bmp)|*.bmp";
46:             if(dlg.ShowDialog()==DialogResult.OK)
47:             {
48:                 i=Image.FromFile(dlg.FileName);
49:                 Invalidate();
50:             }
51:         }
52:
53:         void OnTrack(object sender, EventArgs e)
54:         {
55:             Invalidate();
56:         }
57:
58:         void OnSize(object sender, EventArgs e)
59:         {
60:             Invalidate();
61:         }
62:
63:         public Form1()
64:         {
65:             this.Paint+=new PaintEventHandler(OnPaint);
66:             this.SizeChanged+=new EventHandler(OnSize);
67:
68:             b=new Button();
69:
70:             b.Click+=new EventHandler(OnClickB);
```

LISTING 3.5.10 Continued

```
71: 
72:         b.Location=new Point(5,5);
73:         b.Size=new Size(60,22);
74:         b.Text="Image...";
75: 
76:         this.Controls.Add(b);
77: 
78:         t=new TrackBar();
79:         t.Location=new Point(100,5);
80:         t.Size=new Size(200,22);
81:         t.Maximum=255;
82:         t.Minimum=0;
83:         t.ValueChanged+=new EventHandler(OnTrack);
84: 
85:         this.Controls.Add(t);
86:     }
87: 
88:     static void Main()
89:     {
90:         Application.Run(new Form1());
91:     }
92: }
93: }
```

Compile this file with the following command line:

```
csc /t:winexe imagealpha.cs
```

The important lines are 34–38, deceptively simple for such a powerful feature. Line 34 creates an identity matrix. Line 35 uses the value in the track bar to update the image alpha component of the matrix. Line 36 creates an `ImageAttributes` class that uses the `ColorMatrix` on line 37. Finally, line 38 paints the image onto the screen, allowing our concentric shape background to show through or not, depending on the track bar setting.

Figure 3.5.14 shows this effect with the track bar set to around 70 percent. In case you’re wondering, the Bike is a 1971 Triumph Bonneville 750.

**FIGURE 3.5.14***Alpha blending a bitmap image.*

3.5

GDI+: THE .NET GRAPHICS INTERFACE

Other Color Space Manipulations

Having a whole huge matrix for alpha blending alone seems like overkill, until you realize that any manipulation of the color space is possible with such a tool, if only you knew what numbers to plumb in. The technique of manipulating such a matrix for the purpose of changing the color space is called *recoloring* and is also easy to do with GDI+. Once again, the tool used is the *ColorMatrix* and the vehicle is the *ImageAttributes*. Unfortunately, there are no methods on the *ColorMatrix* to perform color space rotations, but they would be possible, given the correct settings of the linear part of the matrix. As an exercise, you could send in a modification of Listing 3.5.10 to sams@netedgesoftware.com that does color space rotations. The best 10 correct ones within the first year of publication of this book get a free NetEdge Software Polo shirt. To get you started, Listing 3.5.11 is a modification of the previous that allows you to set the red, green, and blue component levels of your chosen image.

LISTING 3.5.11 ColorSpace1.cs: More Color Space Transformations

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Drawing.Imaging;
5: using System.Collections;
6: using System.ComponentModel;
7: using System.Windows.Forms;
8: using System.Data;
9:
10: namespace ColorSpace1
```

LISTING 3.5.11 Continued

```
11: {
12:     class Form1 : Form
13:     {
14:
15:         Button b;
16:         TrackBar tr,tg,tb;
17:         Image i;
18:
19:         void OnPaint(object Sender,PaintEventArgs e)
20:         {
21:             SolidBrush b=new SolidBrush(Color.Red);
22:             Rectangle r=this.ClientRectangle;
23:             GraphicsPath pth=new GraphicsPath();
24:             if(i!=null)
25:             {
26:                 ColorMatrix m=new ColorMatrix();
27:                 m.Matrix00=(float)(1.0/256*tr.Value);
28:                 m.Matrix11=(float)(1.0/256*tg.Value);
29:                 m.Matrix22=(float)(1.0/256*tb.Value);
30:                 ImageAttributes ia=new ImageAttributes();
31:                 iaSetColorMatrix(m);
32:                 e.Graphics.DrawImage(i,this.ClientRectangle,0,
33:                     0,i.Width,i.Height,GraphicsUnit.Pixel,ia);
34:             }
35:         }
36:
37:         void OnClickB(object sender, EventArgs e)
38:         {
39:             OpenFileDialog dlg=new OpenFileDialog();
40:             dlg.Filter="Bitmap files(*.bmp)|*.bmp";
41:             if(dlg.ShowDialog()==DialogResult.OK)
42:             {
43:                 i=Image.FromFile(dlg.FileName);
44:                 Invalidate();
45:             }
46:         }
47:
48:         void OnTrack(object sender, EventArgs e)
49:         {
50:             Invalidate();
51:         }
52:
53:         void OnSize(object sender, EventArgs e)
54:         {
55:             Invalidate();
```

LISTING 3.5.11 Continued

```
56:      }
57:
58:      public Form1()
59:      {
60:          this.Paint+=new PaintEventHandler(OnPaint);
61:          this.SizeChanged+=new EventHandler(OnSize);
62:
63:          b=new Button();
64:
65:          b.Click+=new EventHandler(OnClickB);
66:
67:          b.Location=new Point(5,5);
68:          b.Size=new Size(60,22);
69:          b.Text="Image...";
70:
71:          this.Controls.Add(b);
72:
73:          tr=new TrackBar();
74:          tr.Location=new Point(100,5);
75:          tr.Size=new Size(200,22);
76:          tr.Maximum=255;
77:          tr.Minimum=0;
78:          tr.ValueChanged+=new EventHandler(OnTrack);
79:
80:
81:          this.Controls.Add(tr);
82:
83:          tg=new TrackBar();
84:          tg.Location=new Point(100,55);
85:          tg.Size=new Size(200,22);
86:          tg.Maximum=255;
87:          tg.Minimum=0;
88:          tg.ValueChanged+=new EventHandler(OnTrack);
89:
90:
91:          this.Controls.Add(tg);
92:
93:          tb=new TrackBar();
94:          tb.Location=new Point(100,105);
95:          tb.Size=new Size(200,22);
96:          tb.Maximum=255;
97:          tb.Minimum=0;
98:          tb.ValueChanged+=new EventHandler(OnTrack);
99:
100:
```

LISTING 3.5.11 Continued

```
101:         this.Controls.Add(tb);  
102:     }  
103:  
104:     static void Main()  
105:     {  
106:         Application.Run(new Form1());  
107:     }  
108: }  
109: }
```

Compile this file with the following command line:

```
csc /t:winexe colorspace1.cs
```

This file is substantially identical to that in Listing 3.5.10 with the exception that track bars for red, green, and blue are employed, instead of for alpha. The matrix is set up on lines 27–29 to adjust the intensity of each of the R, G, and B color channels individually.

Summary

There is so much in GDI+ to explore that this chapter could go on for another hundred pages or so. We think, however, that what is here will give you the confidence to experiment further and will allow you to understand most of the conventions required to transition from the old GDI that we know and love to the exiting possibilities of GDI+. Read on now to discover more about Windows Forms applications and components in the last chapter of this section, “Practical Windows Forms Applications.”

Practical Windows Forms Applications

CHAPTER

3.6

IN THIS CHAPTER

- **Using the Properties and Property Attributes** 376
- **Demonstration Application: FormPaint.exe** 383

This section of the book has covered a lot of ground, from basic Windows Forms principles to using GDI+ in your applications. In this last chapter in this section, we'll explore the practical aspects of creating applications with Windows Forms. A critical part of the .NET framework is the ability of each object to describe itself and for you, the programmer, to access its properties. This capability can be used in your own programs to assist your users and to provide a clean and consistent user interface.

In recent years, the Windows platforms have made increasing use of “properties”. You can usually right-click a item and select Properties from the context menu. The idea of this is a good one but, until now, there has never been a standard and consistent way of displaying and editing these properties in your code. Usually a programmer would re-engineer a property system for every application with different dialogs or windows. Now, .NET provides a simple method for creating and using standardized property interfaces, the property attributes, and the property grid.

Using the Properties and Property Attributes

The simple class shown in Listing 3.6.1 has properties.

LISTING 3.6.1 The SimpleObject Class

```
1: public class SimpleObject
2: {
3:     private int _int;
4:     private string _string;
5:     private Color _color;
6:
7:     public SimpleObject()
8:     {
9:         //
10:        // TODO: Add constructor logic here
11:        //
12:     }
13:
14:     public int TheInteger
15:     {
16:         get
17:         {
18:             return _int;
19:         }
20:         set
21:         {
22:             _int=value;
23:         }
24:     }
```

LISTING 3.6.1 Continued

```
25:  
26:     public string TheString  
27:     {  
28:         get  
29:         {  
30:             return _string;  
31:         }  
32:         set  
33:         {  
34:             _string=value;  
35:         }  
36:     }  
37:  
38:     public Color TheColor  
39:     {  
40:         get  
41:         {  
42:             return _color;  
43:         }  
44:         set  
45:         {  
46:             _color=value;  
47:         }  
48:     }  
49: }
```

3.6

PRACTICAL
WINDOWS FORMS
APPLICATIONS

You can see that the class has three private data members—an integer, a string, and a color. These all have public accessor properties—TheInteger, TheString, and TheColor. This is a good strategy from a design perspective because it isolates the encapsulated data and gives a clear and unambiguous interface.

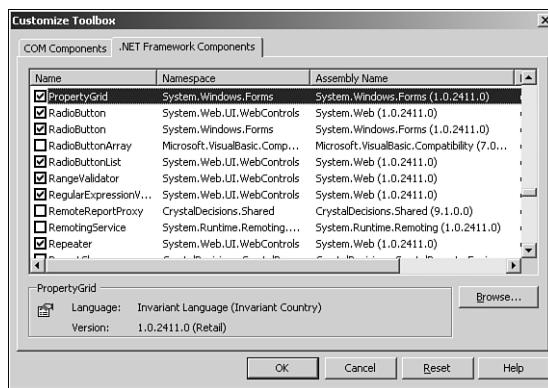
Using our object in an application is fairly simple. What follows is a walk-through sequence for Visual Studio.NET.

Create a new C# Windows application project.

NOTE

It's interesting to note that one of the most useful components in the toolbox—the property grid—isn't shown by default, you have to add it to the list manually. To update the toolbox, right-click it and select Customize Toolbox, you'll see the dialog shown in Figure 3.6.1.

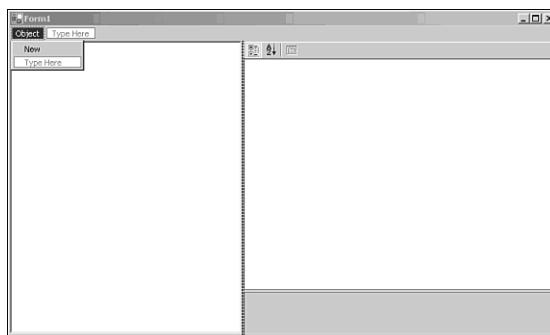
Under the .NET Framework Components tab, select the Property Grid and ensure that its box is checked.

**FIGURE 3.6.1***Customizing the toolbox.*

Returning to the application, it's time to place some controls.

1. Drag a **ListBox** from the toolbox onto the form. Set its **Dock** property to **Left**.
2. Drag a **Splitter** onto the form, it will automatically dock to the rightmost edge of the **ListBox**.
3. Drag a **PropertyGrid** control to the form and place it to the right of the splitter bar. Set its **Dock** property to **Fill**.
4. Drag a **MainMenu** onto the form and place it on the toolbar, call it **Object** and make the first menu entry **New**.

At this point, you should have a form that looks like the one seen in Figure 3.6.2.

**FIGURE 3.6.2***The newly created Object form with Listbox, Splitter, and PropertyGrid.*

Go to the project in the Solution Explorer. Right-click it and select Add New Item. Select C# class from the wizard menu (indicated by the following icon).



Create the `SimpleObject` class, as shown in Listing 3.6.1. Be sure to duplicate it exactly as shown.

Now, to add a handler for the Object, New menu item, double-click the menu entry in the form designer, right where it says New. You'll be taken to the place in the code where the editor has added a handler for you. Fill out the handler as shown in Listing 3.6.2

LISTING 3.6.2 The New Object Menu Item Handler

```
1: private void menuItem2_Click(object sender, System.EventArgs e)
2: {
3:     SimpleObject o=new SimpleObject();
4:     this.listBox1.Items.Add(o);
5: }
```

Finally, create a handler for the `ListBox`'s

`SelectedIndexChanged` event by selecting the `ListBox` in the Designer, clicking the Events icon in the Property Browser, and clicking the `SelectedIndexChanged` entry. Again, you'll be taken to the code where a handler entry will have been provided. Edit this handler entry so that it's the same as shown in Listing 3.6.3

3.6

PRACTICAL
WINDOWS FORMS
APPLICATIONS

LISTING 3.6.3 The `SelectedIndexChanged` Handler

```
1: private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e)
2: {
3:     this.propertyGrid1.SelectedObject=this.listBox1.SelectedItem;
4: }
```

Now you're ready to run the program. Press F5 and wait for compilation to complete.

You can add as many `SimpleObjects` to the list box as you want using the menu entry. After added, selecting a menu entry in the list box will display its properties in the `PropertyGrid` control to the right of the form. Notice how the property grid assists you in editing the properties. The `Color` property is especially interesting because it allows you to select from a palette of colors, enter a known color name (for example, Red), or type in the discrete values for the red, green and blue components, such as 90,180,240. Figure 3.6.3 shows the application in action.

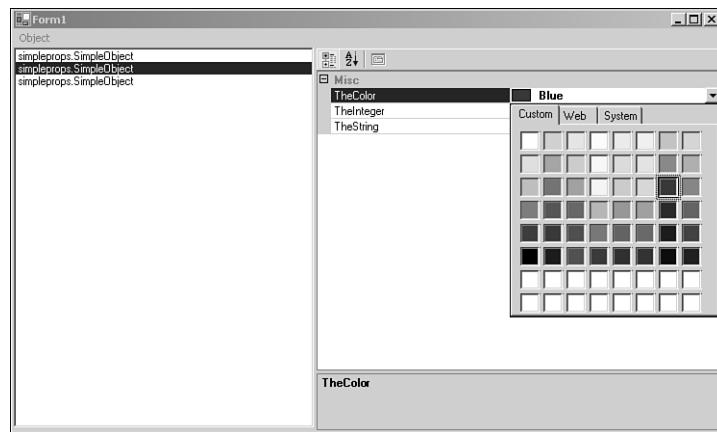


FIGURE 3.6.3

The Property browser application.

For such a small amount of work, this is a powerful application indeed.

Enhancing the Property Experience

If you look carefully at the image in Figure 3.6.3, you'll see a couple of interesting things provided by the framework and the property grid. First, the feedback box at the bottom of the property grid has some text in it. This is feedback to help the user understand what they're doing. Second, the list of properties in the panel is grouped into a category called Misc. If you look at vs.NET or other Windows Forms applications, you might see other categories, such as Behavior or Appearance.

You can arrange your properties into categories and make them provide feedback by using the attributes provided by the framework. The particular classes used are `CategoryAttribute` and the `DescriptionAttribute`. When used in your code, the name is shortened to `Category` or `Description`. The compiler will add the `Attribute` part for you.

Let's revisit the original `SimpleObject` class again. The snippet of code that follows shows these attributes added to one of the properties.

```
[Category("Appearance")]
[Description("controls the color of your moods")]
public Color TheColor
{
    get
    {
        return _color;
    }
```

```
set
{
    _color=value;
}
}
```

When this property is viewed in the Property Grid, it will be sorted into the Appearance category, and the description will be displayed in the feedback panel at the bottom of the Property Grid pane.

The class can also have attributes that help you set up the Property Grid. The [DefaultProperty("propertyname")] attribute selects which of the properties is displayed first in the Property Grid. For example, see line 12 of Listing 3.6.4.

LISTING 3.6.4 Adding Attributes to SimpleObject

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Windows.Forms.Design;
5: using System.ComponentModel;
6:
7: namespace simpleprops
8: {
9:     /// <summary>
10:    /// Summary description for SimpleObject.
11:    /// </summary>
12:    [DefaultProperty("TheInteger")]
13:    public class SimpleObject
14:    {
15:        private int _int;
16:        private string _string;
17:        private Color _color;
18:
19:        public SimpleObject()
20:        {
21:            //
22:            // TODO: Add constructor logic here
23:            //
24:        }
25:
26:
27:        [Category("Nothing Special")]
28:        [Description("Yes guy's 'n gal's, its an integer")]
29:        public int TheInteger
30:        {
```

LISTING 3.6.4 Continued

```
31:         get
32:         {
33:             return _int;
34:         }
35:         set
36:         {
37:             _int=value;
38:         }
39:     }
40:
41:     [Category("A peice of string")]
42:     [Description("This string does absolutely nothing but tie up your
➥time")]
43:     public string TheString
44:     {
45:         get
46:         {
47:             return _string;
48:         }
49:         set
50:         {
51:             _string=value;
52:         }
53:     }
54:
55:     [Category("Appearance")]
56:     [Description("controls the color of your moods")]
57:     public Color TheColor
58:     {
59:         get
60:         {
61:             return _color;
62:         }
63:         set
64:         {
65:             _color=value;
66:         }
67:     }
68: }
69: }
```

Attributes can also be used to prevent the property from being seen in the browser. This is useful if you want to declare a public property for the programmer but keep it from the user. To do this, use the `[Browsable(false)]` attribute.

Other attributes for properties are only useful if you're writing controls to be used in a design environment, such as VS.NET. The creation of such controls is not within the scope of this book.

Demonstration Application: **FormPaint.exe**

This application puts together many of the principals we've shown you in this book and shows off some of the features of Windows Forms in a complete application context.

Form Paint is a Multiple Document Interface (MDI) application that allows you to paint images with brushes and shapes. It has customized menus and shows off some of the power of GDI+ also.

Part 1: The Basic Framework

Getting started, create an application in VS.NET and immediately set the forms `IsMDIContainer` to true. This makes it a main parent window for other form-based documents that are hosted in it.

The name, `Form1`, must be changed to `MainForm` and the title of the window changed to `FormPaint`. Remember that when the name of the `MainForm` changes, you must also manually modify the static `Main` method as follows:

```
[STAThread]
static void Main()
{
    Application.Run(new MainForm()); // change the application name
}
```

Drag a `MainMenu` from the toolbox onto the `MainForm`. Type the word `&File` as the menu name and add three menu items—`&New`, `&Open`, and `&Exit`.

Drag an `OpenFileDialog` and a `SaveFileDialog` from the toolbox onto the `MainForm`. These will show up in the icon tray below the form.

Double-click the `Open` menu item in the main menu and type the following into the handler provided:

```
this.openFileDialog1.Filter = "Bitmap files (*.BMP) | *.bmp";
this.openFileDialog1.ShowDialog();
```

Create a second menu called `Windows`. This will be used to keep track of the MDI child forms in the application. This is done automatically for you if you set the `MDIList` property of the menu to true.

Next, add a new Windows Form to the project, call it `ChildForm`. Set the `BackColor` to green.

Add a private `Image` variable called `myImage` to the child form and a public accessor property as follows:

```
private Image myImage;
public Image Image
{
    set{myImage =value;}
    get{return myImage;}
}
```

Now, double-click the OpenFileDialog in the icon tray. This action will create a `FileOk` handler for you. Type the following into the resulting handler:

```
ChildForm c=new ChildForm();
c.MdiParent=this;
c.Image=Image.FromFile(this.openFileDialog1.FileName);
c.Text=this.openFileDialog1.FileName;
c.Show();
```

This will open a file, create a child form, and load the image into the child form. The child form now needs to display the image. Select the `ChildForm` design page, click the Events button in the Property Browser, and double-click the Paint event. This creates a `PaintEventHandler` delegate for the `ChildForm` and opens the editor at the correct place in the file. Type the following into the handler:

```
e.Graphics.DrawImage(myImage,0,0,myImage.Width,myImage.Height);
```

This is a good place to stop and take stock of the project. Compile and run the code using the F5 key. You'll see an application that lets you load and display images in multiple windows. Figure 3.6.4 shows this basic application in action.

This portion of the program is available as FormPaint Step1 on the Sams Web site associated with this book.

Part 2: Scrolling a Window and Creating New Images

If you've run the fledgling `FormPaint` program and loaded up a few images, you'll have noticed that the windows will resize. But, when they become too small to display the picture, they simply chop it off without giving you a chance to see the sides with a scrollbar. We'll correct this quickly by adding some simple code to the `ChildForm`.



FIGURE 3.6.4
The basic MDI application.

3.6

PRACTICAL WINDOWS FORMS APPLICATIONS

First, in the `Image` property set accessor for the `ChildForm`, we'll add a command to set the `AutoScrollMinSize` property to the size of the image in `myImage`. The following code snippet shows the added functionality on line 6.

```
1: public Image Image
2: {
3:     set
4:     {
5:         myImage = value;
6:         this.AutoScrollMinSize = myImage.Size;
7:     }
8:     get { return myImage; }
9: }
```

Now, whenever the client rectangle size drops below the minimum size in either dimension, the corresponding scrollbar will appear.

A second change must be made to the `OnPaint` handler to position the scrollbars. Here, we need to offset the image by the value of the scrollbar position so that the image is painted correctly. The form also provides a property for the scrollbar positions that is shown in the following code snippet.

```
1: private void ChildForm_Paint(object sender,
2:   System.Windows.Forms.PaintEventArgs e)
3: {
4:   e.Graphics.DrawImage(myImage,
5:     this.AutoScrollPosition.X,
6:     this.AutoScrollPosition.Y,
7:     myImage.Width,
8:     myImage.Height);
8: }
```

On lines 4 and 5, you can see that the X and Y values of the form's AutoScrollPosition property offset the origin of the image by the correct amount.

So far, the program can only load an image that exists on disk. We need to create an image, perhaps with a choice of sizes, and to allow that image to be saved to disk also.

Creating the image can be accomplished as follows. Using the MainForm design page, select and double-click the New menu entry in the File menu. This will create a handler that we'll fill in later.

Now, right-click the FormPaint project and choose Add, New Item. Choose a new Form and call it **NewImageDialog.cs**. Drag three radio buttons and two buttons from the toolbox onto the dialog. Then arrange them, as shown in Figure 3.6.5

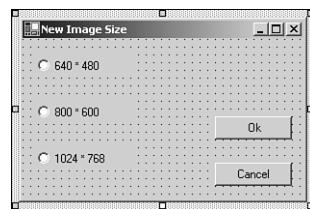


FIGURE 3.6.5
Button arrangement dialog.

The DialogResult properties of the OK and Cancel buttons must be set to DialogResult.OK and DialogResult.Cancel respectively. To retrieve the setting from the radio buttons, we can add a simple property that returns a size. Hand-edit the **NewImageDialog.cs** code and add the following public property:

```
public Size ImageSize
{
    get
    {
        if(this.radioButton2.Checked)
            return new Size(800,600);
```

```

    if(this.radioButton3.Checked)
        return new Size(1024,768);
    return new Size(640,480);
}
}

```

Now we can return to the handler we created earlier and fill it out to create a blank image.

Find the handler in the `MainForm.cs` file and add the necessary code as follows:

```

1: private void menuItem2_Click(object sender, System.EventArgs e)
2: {
3:     // To create a new file...
4:     NewImageDialog dlg = new NewImageDialog();
5:     if(dlg.ShowDialog()==DialogResult.OK)
6:     {
7:         ChildForm c=new ChildForm();
8:         c.MdiParent=this;
9:         c.Image=new Bitmap(dlg.ImageSize.Width,dlg.ImageSize.Height);
10:        Graphics g=Graphics.FromImage(c.Image);
11:        g.FillRectangle(new SolidBrush(Color.White),0,0,
12:                      c.Image.Width,c.Image.Height);
12:        c.Show();
13:    }
14: }

```

3.6

Adding the Save Functions

Saving the file or saving “as” a different file will be performed through the main File menu. Add a Save and Save As menu item, remembering to use the ampersand before the key-select characters—for example, **&Save** and **Save &As**. For esthetic reasons, you might also want to move these into a sensible place on the menu and place a separator between the `file` and `exit` functions.

When you’ve created these entries, go ahead and double-click each one in turn to create the handlers for them. The following code snippet shows the two save handlers:

```

1: private void menuItem6_Click(object sender, System.EventArgs e)
2: {
3:     //Save the image file.
4:     ChildForm child = (ChildForm)this.ActiveMdiChild;
5:     child.Image.Save(child.Text);
6: }
7:
8: private void menuItem7_Click(object sender, System.EventArgs e)
9: {
10:    //Save the image file as a different filename
11:    ChildForm child = (ChildForm)this.ActiveMdiChild;
12:    this.saveFileDialog1.FileName=child.Text;

```

```
13:    if(this.saveFileDialog1.ShowDialog()==DialogResult.OK)
14:    {
15:        child.Image.Save(this.saveFileDialog1.FileName);
16:        child.Text=this.saveFileDialog1.FileName;
17:    }
18: }
```

Lines 1–6 are the simple save handler. The filename is already known, so a save is performed using the titlebar text of the `ChildForm`.

Lines 8–18 use this filename as a starting point but invoke the `SaveFileDialog` to allow the user to choose a new name.

This behavior is fine if you can guarantee that a `ChildWindow` is always open from which to get the text. If you use the handlers on an empty main form, though, an exception will occur because there is no active MDI child. This problem can be overcome in one of two ways. Either write an exception handling `try...catch` block into both handlers or, alternatively, never allow the handler to be called if there are no MDI children. The second method is best for the purposes of this demonstration because it allows us to use the menu pop-up events correctly.

You might remember from Chapter 3.1, “Introduction to Windows Forms,” that the menu `Popup` event is fired when the user selects the top-level menu on the toolbar just before the menu is drawn. This gives us the chance to modify the menu behavior to suit the current circumstances. The following code snippet shows the `Popup` handler for the `File` menu item.

```
1: private void menuItem1_Popup(object sender, System.EventArgs e)
2: {
3:     if(this.MdiChildren.Length!=0)
4:     {
5:         menuItem6.Enabled=true;
6:         menuItem7.Enabled=true;
7:     }
8:     else
9:     {
10:         menuItem6.Enabled=false;
11:         menuItem7.Enabled=false;
12:     }
13: }
```

The handler checks to see if there are any MDI children in the main form (line 3). If there are, it enables both the save menus on lines 5 and 6. If there are none, it disables these menus on lines 10 and 11.

This part of the application is available as `FormPaint Step2` on the Sams Web site associated with this book.

Part 3: More User Interface

With the image loading, display, and saving in place, we can get on with the task of adding the rest of the user interface items that will be used to drive the program. We need a tool palette; in this case, a simple one will suffice to demonstrate principles, a status bar, some mouse handling, and a custom button class based on `UserControl`.

Creating a Custom User Control

A big advantage to user controls in .NET is their ease of reuse. A quick side-trip into a custom control project can result in a handy component that you'll use many times in various projects. These controls play nicely in the design environment and can be shipped as separate assemblies if you want.

To begin with, add a new C# control library project to the current solution. Remember to choose the Add to Current Solution radio button on the wizard dialog. Call the project **FormPaintControl**. You'll be presented with a wizard to choose the type of control to add. Select a new `UserControl` and name it **CustomImageButton.cs**.

Listing 3.6.5 shows the full source code of the `CustomImageButton` control.

3.6

PRACTICAL
WINDOWS FORMS
APPLICATIONS

LISTING 3.6.5 `CustomImageButton.cs`: The Custom Image Button Class

```
1: using System;
2: using System.Collections;
3: using System.ComponentModel;
4: using System.Drawing;
5: using System.Drawing.Drawing2D;
6: using System.Data;
7: using System.Windows.Forms;
8: using System.Drawing.Imaging;
9:
10: namespace FormPaintControl
11: {
12:     /// <summary>
13:     /// Summary description for CustomImageButton.
14:     /// </summary>
15:     public class CustomImageButton : System.Windows.Forms.UserControl
16:     {
17:         private Image image;
18:         private Color transparentColor;
19:         private bool ownerDraw;
20:         private bool down;
21:
22:         [
23:             Category("Behavior"),
```

LISTING 3.6.5 Continued

```
24:         Description("Allows external paint delegates to draw he control")
25:     ]
26:     public bool OwnerDraw
27:     {
28:         get
29:         {
30:             return ownerDraw;
31:         }
32:         set
33:         {
34:             ownerDraw=value;
35:         }
36:     }
37:
38:     [
39:     Category("Appearance"),
40:     Description("The image displayed on the control")
41:     ]
42:     public Image Image
43:     {
44:         get
45:         {
46:             return image;
47:         }
48:         set
49:         {
50:             image=value;
51:         }
52:     }
53:
54:     [
55:     Category("Appearance"),
56:     Description("The transparent color used in the image")
57:     ]
58:     public Color TransparentColor
59:     {
60:         get
61:         {
62:             return transparentColor;
63:         }
64:         set
65:         {
66:             transparentColor=value;
67:         }
68:     }
```

LISTING 3.6.5 Continued

```
68:      }
69:
70:      protected override void OnSizeChanged(EventArgs e)
71:      {
72:          if(DesignMode)
73:              Invalidate();
74:          base.OnSizeChanged(e);
75:      }
76:
77:      public void DrawFocusRect(Graphics g)
78:      {
79:          Pen p=new Pen(Color.Black,1);
80:          p.DashStyle=DashStyle.Dash;
81:          Rectangle rc=ClientRect;
82:          rc.Inflate(-1,-1);
83:          g.DrawRectangle(p,rc);
84:      }
85:
86:      protected override void OnLostFocus(EventArgs e)
87:      {
88:          Invalidate();
89:          base.OnLostFocus(e);
90:      }
91:
92:      protected override void OnPaint(PaintEventArgs e)
93:      {
94:          Rectangle rc=ClientRect;
95:          rc.Offset(4,4);
96:          e.Graphics.SetClip(ClientRectangle);
97:          if(!ownerDraw)
98:          {
99:              if(image==null || !(image is Bitmap))
100:              {
101:                  Pen p=new Pen(Color.Red,2);
102:                  e.Graphics.DrawRectangle(p,ClientRect);
103:                  e.Graphics.DrawLine(p,ClientRect.Location.X,
104:                      ClientRectangle.Location.Y,
105:                      ClientRectangle.Location.X+ClientRectangle.Width,
106:                      ClientRectangle.Location.Y+ClientRectangle.Height);
107:                  e.Graphics.DrawLine(p,
108:                      ClientRectangle.Location.X+ClientRectangle.Width,
109:                      ClientRectangle.Location.Y,
110:                      ClientRectangle.Location.X,
111:                      ClientRectangle.Location.Y+ClientRectangle.Height);
```

LISTING 3.6.5 Continued

```
112:         return;
113:     }
114:     Bitmap bm=(Bitmap)image;
115:     bm.MakeTransparent(transparentColor);
116:     if(down || Focused)
117:     {
118:         e.Graphics.DrawImage(bm,
119:             rc.Location.X,
120:             rc.Location.Y,
121:             bm.Width,bm.Height);
122:         if(Focused)
123:             DrawFocusRect(e.Graphics);
124:     }
125:     else
126:     {
127:         ControlPaint.DrawImageDisabled(e.Graphics,bm,
128:             rc.Location.X,rc.Location.Y,BackColor);
129:         e.Graphics.DrawImage(bm,
130:             ClientRectangle.Location.X,
131:             ClientRectangle.Location.Y,
132:             bm.Width,bm.Height);
133:     }
134: }
135: base.OnPaint(e);
136: }
137:
138: /// <summary>
139: /// Required designer variable.
140: /// </summary>
141: private System.ComponentModel.Container components = null;
142:
143: public CustomImageButton()
144: {
145:     // This call is required by the Windows.Forms Form Designer.
146:     InitializeComponent();
147:
148:     // TODO: Add any initialization after the InitializeComponent call
149:
150: }
151:
152: protected override void OnMouseEnter(EventArgs e)
153: {
154:     base.OnMouseEnter(e);
155:     down=true;
156:     Invalidate();
```

LISTING 3.6.5 Continued

```
157:      }
158:
159:      protected override void OnMouseLeave(EventArgs e)
160:      {
161:          base.OnMouseLeave(e);
162:          down=false;
163:          Invalidate();
164:      }
165:
166:      [
167:          Browsable(true),
168:          DesignerSerializationVisibility(
169:              ►DesignerSerializationVisibility.Visible)
170:      ]
171:      public override string Text
172:      {
173:          get
174:          {
175:              return base.Text;
176:          }
177:          set
178:          {
179:              base.Text = value;
180:          }
181:
182:          /// <summary>
183:          /// Clean up any resources being used.
184:          /// </summary>
185:          protected override void Dispose( bool disposing )
186:          {
187:              if( disposing )
188:              {
189:                  if( components != null )
190:                      components.Dispose();
191:              }
192:              base.Dispose( disposing );
193:          }
194:
195:          #region Component Designer generated code
196:          /// <summary>
197:          /// Required method for Designer support - do not modify
198:          /// the contents of this method with the code editor.
199:          /// </summary>
```

LISTING 3.6.5 Continued

```
200:     private void InitializeComponent()
201:     {
202:         // 
203:         // CustomImageButton
204:         // 
205:         this.Name = "CustomImageButton";
206:         this.Size = new System.Drawing.Size(64, 56);
207: 
208:     }
209: #endregion
210:
211: }
212: }
```

Beginning with line 1, note the added use of namespaces in the control. The default is only to use the `System` namespace.

Line 10 defines the `FormPaintControl` namespace with line 15 beginning the control itself—`CustomImageButton`.

Lines 17 through 20 define private data members to store information used in the drawing of the control. There is an `Image`, a transparent color key, and two Boolean flags for an owner draw property and to decide whether the button should be rendered down or up.

Public accessor properties for the relevant members ensure that the control will integrate nicely with the design environment and give some user feedback. Notice the `Category` and `Description` attributes preceding the property definitions on lines 26, 42, and 58.

Line 70 defines the `OnSizeChanged` method. When this control is used in Design mode, it needs to paint itself whenever it's resized. It should also call the base class method to ensure that any other delegates added to the `SizeChanged` event will also be called.

Line 77 defines a new method that draws a dotted line around the control when it has the focus. This is called from the `OnPaint` method.

The method on line 86 ensures that the control is redrawn when focus is lost.

The `OnPaint` method, beginning on line 92, has to contend with normal drawing of the control and drawing of the control when it's used in the design environment. This means that in the initial instance, the control will be instantiated by the designer with an empty `image` property. So that exceptions do not occur, the control will draw a red rectangle with a red cross in it if there is no image with which to paint. This is accomplished on lines 102–122. Otherwise, normal drawing takes place on lines 118–123 if the button is down or focused. Lines 127–132

draw the button in its up position. Note the use of the `ControlPaint` method on lines 127 and 128 that draws a “disabled” image behind the main bitmap as a drop shadow. This method also calls the base `OnPaint` method to ensure correct painting when other paint handler delegates are added to the `Paint` event.

The standard control added all lines from 138–150, and our additions continue with the `MouseEnter` and `MouseLeave` handlers on lines 152 and 159, respectively. These make the button depress whenever the mouse floats over it.

On line 167, there is an attribute to allow the `Text` property to be seen in the property browser. The `UserControl` from which this class is derived hides this property, so we need to override it (lines 170–180) and make it visible in the browser. Notice that the implementation calls the base class explicitly. On line 169, the `DesignerSerializationVisibility` attribute is used to ensure that the text is stored in the `InitializeComponent` method.

The rest of the file is standard stuff added by the control wizard and includes a `Dispose` method and an `InitializeComponent` method.

Compile the control and then right-click the toolbox, select Customize Toolbox and you’ll see the dialog shown in Figure 3.6.6.

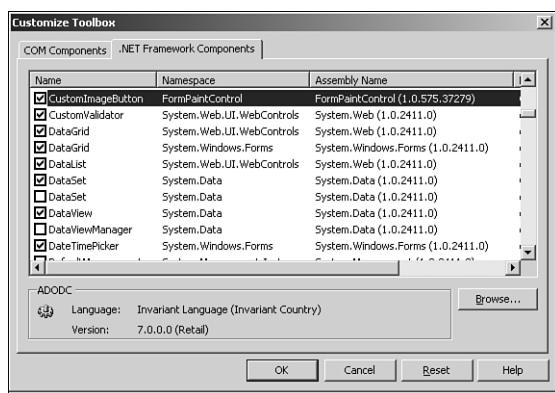


FIGURE 3.6.6

Adding the control to the toolbox.

To see the `CustomImageButton`, you must first find the compiled DLL that contains the control. This could be

```
<your Projects>\FormPaint\FormPaintControl\Debug\Bin\FormPaintControl.dll
```

Ensure that the check box is ticked and the control will appear in your toolbox.

NOTE

You can make this a permanent tool to use in your programs if you change to Release mode, compile the control, and then select the release version of the FormPaintControl DLL. You might also want to copy the release DLL to a place on your hard disk where it's unlikely to be erased.

Using the CustomImageControl

To the `MainForm`, add a panel and dock it to the right side of the form. This area will be used to host our simple tool palette. Drag a status bar from the toolbox onto the form, select the Panels collection in the property browser, and click the button in the combo-box that will appear.

You'll see a dialog that will allow you to add one or more `StatusbarPanel` objects; for now, just add one. Remember to set the `Statusbar` objects `ShowPanels` property to true.

Drag four `CustomImageControl` objects onto the panel and arrange them vertically to make a place for selecting drawing tools. Initially, each will be displayed as a red rectangle with a cross in it.

Using the property browser, add images to each of the buttons' `Image` properties. We selected a paintbrush, rectangle, ellipse, and eraser image for our simple tool palette. Figure 3.6.7 shows these four images. Note that the outside, unused portion of the image will be color-keyed magenta to use as a transparency key.

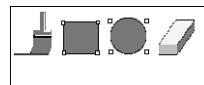


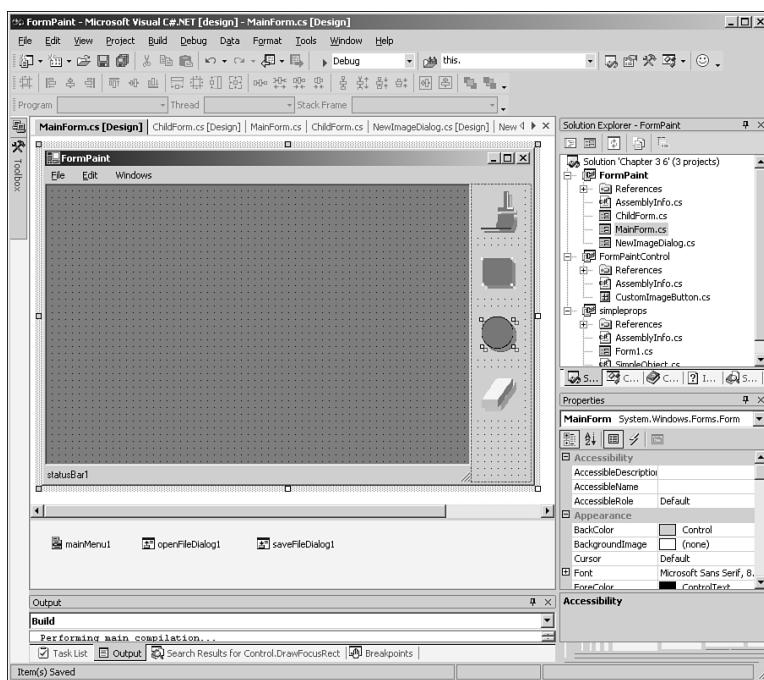
FIGURE 3.6.7

The bitmap images used for the buttons.

When four images, `brush.bmp`, `rect.bmp`, `ellipse.bmp`, and `eraser.bmp` are added to their buttons, the `Paint` method will display them for us, even in the Designer.

The design should look similar to the one in Figure 3.6.8.

The `CustomImageControl` can hold text for us so we can use this to create a simple UI integration and feedback mechanism. We'll store information in the text to annotate the button with tooltips and status bar feedback. Later, we'll use the same feedback mechanism to enhance the menus.

**FIGURE 3.6.8**

The MainForm final design layout.

In the **Text** property of the four buttons, the following text will provide more than just a button caption. The text will contain a caption, tooltip, and status bar text separated by the | vertical line (character 124). For an example, the following line shows the format.

Caption Text|Status Bar feedback|Tooltip text

For each text property in turn, add the relevant line using the property browser:

```
CustomImageButton1.Text = "|Apply color with a brush tool|Paintbrush"
CustomImageButton2.Text = "|Draw a rectangle|Rectangle"
CustomImageButton3.Text = "|Draw an ellipse|Ellipse"
CustomImageButton4.Text = "|Erase an area|Eraser"
```

Note that none of the text entries have a caption portion, because a text caption is unnecessary.

Status and Tooltips

A tooltip gives added useful feedback to the user and our tool palette items have text that they can display when the mouse hovers over them, so the `FormPaint` program uses a simple method to display tooltip and status bar text. Taking advantage of the `String` classes' ability to split strings at given delimiters, we can create a simple class that will extract the tooltip, status bar, and caption text for us. Add a new C# class to the project and call it `CSTSplit` for caption–status–tooltip splitter.

The `CSTSplit` class is simple and is shown in Listing 3.6.6.

LISTING 3.6.6 `CSTSplit.cs`: The Caption–Status–Tooltip Splitter

```
1: using System;
2:
3: namespace FormPaint
4: {
5:     /// <summary>
6:     /// CSTSplit divides up a given string into
7:     /// pieces at a "|" delimiter to provide
8:     /// Caption, Status, and Tooltip text.
9:     /// </summary>
10:    public class CSTSplit
11:    {
12:        string[] splitStrings;
13:
14:        public CSTSplit(string toSplit)
15:        {
16:            splitStrings = toSplit.Split(new char[]{ '|'});
17:        }
18:
19:        public string Caption
20:        {
21:            get
22:            {
23:                if(splitStrings.Length>0)
24:                    return splitStrings[0];
25:                return "";
26:            }
27:        }
28:
29:        public string Status
30:        {
31:            get
32:            {
33:                if(splitStrings.Length>1)
```

LISTING 3.6.6 Continued

```
34:             return splitStrings[1];
35:             return "";
36:         }
37:     }
38:
39:     public string Tooltip
40:     {
41:         get
42:         {
43:             if(splitStrings.Length>2)
44:                 return splitStrings[2];
45:             return "";
46:         }
47:     }
48: }
49: }
```

The constructor on line 10 splits the string provided into an array of up to three strings divided at the vertical line delimiters.

The `Caption`, `Status`, and `Tooltip` properties (lines 19, 29, and 39, respectively) retrieve the correct string or provide a blank if no text for that portion of the string was included.

The initial use for this class will be to add tooltips to the four buttons in the tool palette. This is accomplished in the constructor of the `MainForm`, just after the `InitializeComponent` call. To display the tooltips, drag a `ToolTip` object from the toolbox onto the `MainForm` design page. The `ToolTip` will appear in the icon tray below the main window. Then add the following code to the `MainForm` constructor:

```
CSTSsplit splitter=new CSTSplit(this.customImageButton1.Text);
this.toolTip1.SetToolTip(this.customImageButton1,splitter.Tooltip);
splitter=new CSTSplit(this.customImageButton2.Text);
this.toolTip1.SetToolTip(this.customImageButton2,splitter.Tooltip);
splitter=new CSTSplit(this.customImageButton3.Text);
this.toolTip1.SetToolTip(this.customImageButton3,splitter.Tooltip);
splitter=new CSTSplit(this.customImageButton4.Text);
this.toolTip1.SetToolTip(this.customImageButton4,splitter.Tooltip);
```

You can see that each `CustomImageButton` in turn is interrogated for its text, and the `Tooltip` property from the `CSTSsplit` object returns the correct text that is handed to the `ToolTip`'s `ShowTip` method.

Now, when the mouse rests on the control for half a second or more, a tooltip will be displayed containing the correct text.

Each of these controls contain text for the status bar also. To make this text show itself, select the first **CustomImageButton** control in the designer and type the method name **ShowStatus** into the **MouseEnter** event in the property browser. A handler will be created that should be filled out as follows.

```
private void ShowStatus(object sender, System.EventArgs e)
{
    Control c=(Control)sender;
    CSTSplit splitter=new CSTSplit(c.Text);
    this.statusBarPanel1.Text=splitter.Status;
}
```

Now, compiling and running the program will show that the tooltips and status bar messages are functioning correctly.

This version of the code is saved as **FormPaint Step3**.

Part 4: Tool Properties and Application

The program is now in a state where we can begin to do real work. We need to be able to select individual properties for the tools and to apply them to the bitmap image.

The **Paintbrush** object requires several properties, such as brush shape, color, and size. The ellipse and rectangle tool only need to define line color, fill color, and line thickness. The eraser will have a size and shape, like the paintbrush, but will always erase to white.

So that we can use the **PropertyGrid** to select these properties, we'll create a class for each of these tools that will be used to retain the user selections.

Defining the Tool Properties

The three tool description objects are shown in Listings 3.6.7, 3.6.8 and 3.6.9.

LISTING 3.6.7 PaintBrushProperties.cs: The Paintbrush Tool Properties

```
1: using System;
2: using System.ComponentModel;
3: using System.Drawing;
4: using System.Globalization;
5:
6: namespace FormPaint
7: {
8:
9:     public enum Shape
10:    {
11:        Round,
12:        Square,
```

LISTING 3.6.7 Continued

```
13:     Triangle
14: };
15:
16:
17: public class PaintBrushProperties
18: {
19:     private Shape shape;
20:     private int size;
21:     private Color color;
22:     private int transparency;
23:
24:     public object Clone()
25:     {
26:         return new PaintBrushProperties(shape, size, color, transparency);
27:     }
28:
29:     protected PaintBrushProperties(Shape _shape, int _size,
30:                                     Color _color, int _transparency)
31:     {
32:         shape=_shape;
33:         size=_size;
34:         color=_color;
35:         transparency=_transparency;
36:     }
37:
38:     [
39:         Category("Brush"),
40:         Description("The brush shape")
41:     ]
42:     public Shape Shape
43:     {
44:         get{ return shape; }
45:         set{ shape = value; }
46:     }
47:
48:     [
49:         Category("Brush"),
50:         Description("The brush color")
51:     ]
52:     public Color Color
53:     {
54:         get{return color;}
55:         set{color = value;}
56:     }
```

LISTING 3.6.7 Continued

```
57:  
58:      [  
59:          Category("Brush"),  
60:          Description("The size of the brush in pixels")  
61:      ]  
62:      public int Size  
63:      {  
64:          get{return size;}  
65:          set{size = value;}  
66:      }  
67:  
68:      [  
69:          Category("Brush"),  
70:          Description("Percentage of transparency for the brush")  
71:      ]  
72:      public int Transparency  
73:      {  
74:          get{return transparency;}  
75:          set{transparency = value;}  
76:      }  
77:  
78:      public PaintBrushProperties()  
79:      {  
80:          color=Color.Black;  
81:          size=5;  
82:          shape=Shape.Round;  
83:          transparency = 0;  
84:      }  
85:  }  
86: }
```

NOTE

Notice the Shape enumeration at the beginning of this file.

LISTING 3.6.8 ShapeProperties.cs: The Shape Properties Class

```
1: using System.Drawing;  
2:  
3: namespace FormPaint  
4: {  
5:     public class ShapeProperties
```

LISTING 3.6.8 Continued

```
6:      {
7:          private Color fillColor;
8:          private Color lineColor;
9:          private bool line;
10:         private bool fill;
11:         private int lineWidth;
12:
13:         public object Clone()
14:         {
15:             return new ShapeProperties(fillColor,lineColor,
16:                                         line,fill,lineWidth);
17:         }
18:
19:         protected ShapeProperties(Color _fillColor, Color _lineColor,
20:                                   bool _line, bool _fill, int _lineWidth)
21:         {
22:             fillColor = _fillColor;
23:             lineColor = _lineColor;
24:             fill = _fill;
25:             line = _line;
26:             lineWidth = _lineWidth;
27:         }
28:
29:
30:         [
31:             Category("Geometric Shape"),
32:             Description("The color to fill the shape with")
33:         ]
34:         public Color FillColor
35:         {
36:             get{return fillColor;}
37:             set{fillColor=value;}
38:         }
39:
40:         [
41:             Category("Geometric Shape"),
42:             Description("The color to outline the shape with")
43:         ]
44:         public Color LineColor
45:         {
46:             get{return lineColor;}
47:             set{lineColor=value;}
48:         }
49:
```

LISTING 3.6.8 Continued

```
50:      [
51:          Category("Geometric Shape"),
52:          Description("The width of the line")
53:      ]
54:      public int LineWidth
55:      {
56:          get{return lineWidth;}
57:          set{lineWidth=value;}
58:      }
59:
60:      [
61:          Category("Geometric Shape"),
62:          Description("Draw the outline")
63:      ]
64:      public bool Line
65:      {
66:          get{return line;}
67:          set{line = value;}
68:      }
69:
70:      [
71:          Category("Geometric Shape"),
72:          Description("Fill the shape")
73:      ]
74:      public bool Fill
75:      {
76:          get{return fill;}
77:          set{fill = value;}
78:      }
79:
80:      public ShapeProperties()
81:      {
82:      }
83:  }
84: }
```

LISTING 3.6.9 EraserProperties.cs: The Eraser Properties Class

```
1: using System;
2: using System.ComponentModel;
3:
4: namespace FormPaint
5: {
6:     public class EraserProperties
```

LISTING 3.6.9 Continued

```
7:      {
8:          private Shape shape;
9:          private int size;
10:
11:         public object Clone()
12:         {
13:             return new EraserProperties(shape,size);
14:         }
15:
16:         protected EraserProperties(Shape _shape, int _size)
17:         {
18:             shape=_shape;
19:             size=_size;
20:         }
21:
22:         [
23:             Category("Eraser"),
24:             Description("The Eraser shape")
25:         ]
26:         public Shape Shape
27:         {
28:             get{ return shape; }
29:             set{ shape = value; }
30:         }
31:
32:         [
33:             Category("Eraser"),
34:             Description("The size of the Eraser in pixels")
35:         ]
36:         public int Size
37:         {
38:             get{return size;}
39:             set{size = value;}
40:         }
41:
42:         public EraserProperties()
43:         {
44:             shape=Shape.Square;
45:             size=10;
46:         }
47:     }
48: }
```

Notice that all these objects have a `Clone()` method. This is used when the properties are edited in a dialog box, which we'll define next, and the user has the option to click the Cancel button. If the dialog is cancelled, the clone of the object is discarded.

Editing the Tool Properties

The editing for all tool properties is accomplished with the same dialog box. This simple form hosts a property grid that deals with all the editing of the properties within the classes.

This dialog, `ToolProperties`, is created with the IDE as follows.

Drag two buttons from the toolbar to the dialog, label them **Ok** and **Cancel**, and then set their `DialogResult` properties to the corresponding value.

Drag and position a `PropertyGrid` object onto the design surface. The final result should look similar to the image in Figure 3.6.9.

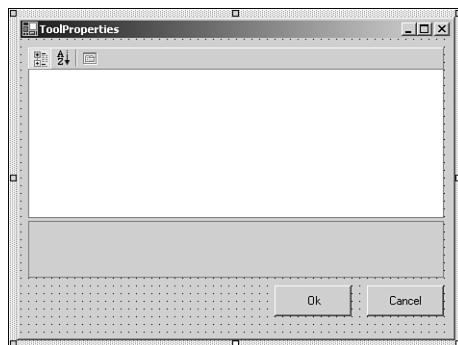


FIGURE 3.6.9

The ToolProperties dialog.

This dialog needs some added behavior, so we'll add a property to set and retrieve the object that the property grid edits. The following code snippet shows this simple accessor:

```
public object Object
{
    get{return this.propertyGrid1.SelectedObject;}
    set{this.propertyGrid1.SelectedObject=value;}
}
```

After this accessor is in place, we can add the functionality for editing and using the properties.

The `MainForm` must be modified to hold the properties and the tool currently in use. The code snippets that follow show these additions.

The Tool structure is added to the FormPaint namespace and is used to define which tool is in use:

```
public enum Tool
{
    Paintbrush,
    Rectangle,
    Ellipse,
    Eraser
}
```

The following private members are added to the MainForm class:

```
private Tool currentTool;
private PaintBrushProperties paintbrushProperties;
private ShapeProperties shapeProperties;
private EraserProperties eraserProperties;
```

The following additions are made to the MainForm constructor:

```
paintbrushProperties = new PaintBrushProperties();
shapeProperties = new ShapeProperties();
eraserProperties = new EraserProperties();
```

3.6

Adding Handlers for the Tool Properties

To use a tool, the user will single click it. To edit the properties for a tool, the user will double-click. To do this, add a click handler to each of the four tool buttons on the main form. The following code snippet shows how to fill these out:

```
private void ClickPaintbrush(object sender, System.EventArgs e)
{
    currentTool=Tool.Paintbrush;
}

private void ClickRectangle(object sender, System.EventArgs e)
{
    currentTool=Tool.Rectangle;
}

private void ClickEllipse(object sender, System.EventArgs e)
{
    currentTool=Tool.Ellipse;
}

private void ClickEraser(object sender, System.EventArgs e)
{
    currentTool=Tool.Eraser;
}
```

You can see that these handlers simply change the values of the `currentTool` member in the `MainForm`.

All four buttons now need to be tied to the same `DoubleClick` handler. Begin by creating a handler, called **OnToolProperties**, for one button and filling it out as follows:

```
private void OnToolProperties(object sender, System.EventArgs e)
{
    ToolProperties dlg=new ToolProperties();
    switch(currentTool)
    {
        case Tool.Paintbrush:
            dlg.Object=paintbrushProperties.Clone();
            break;
        case Tool.Rectangle:
            dlg.Object=shapeProperties.Clone();
            break;
        case Tool.Ellipse:
            dlg.Object=shapeProperties.Clone();
            break;
        case Tool.Eraser:
            dlg.Object=eraserProperties.Clone();
            break;
    }
    if(dlg.ShowDialog()==DialogResult.OK)
    {
        switch(currentTool)
        {
            case Tool.Paintbrush:
                paintbrushProperties=(PaintBrushProperties)dlg.Object;
                break;
            case Tool.Rectangle:
                shapeProperties=(ShapeProperties)dlg.Object;
                break;
            case Tool.Ellipse:
                shapeProperties=(ShapeProperties)dlg.Object;
                break;
            case Tool.Eraser:
                eraserProperties=(EraserProperties)dlg.Object;
                break;
        }
    }
}
```

This handler decides which of the property classes to use, hands the correct class to the editor, invokes the editor and then discards or replaces the edited property, depending on the `DialogReturn` value.

To allow the `ChildForm` to access the `MainForm` variables, such as the current drawing tool or the brush and shape properties, we have added some accessor properties. In addition, there is an accessor that allows the `ChildForm` to place text in the `MainForm`'s status bar. This is useful for user feedback. The snippet that follows contains the final additions to the class:

```
public Tool CurrentTool
{
    get{return currentTool;}
}

public ShapeProperties ShapeProperties
{
    get{return shapeProperties;}
}

public EraserProperties EraserProperties
{
    get{return eraserProperties;}
}

public PaintBrushProperties PaintBrushProperties
{
    get{return paintbrushProperties;}
}

public string StatusText
{
    set{this.statusBarPanel1.Text=value;}
}
```

Running the application now will allow you to test the tool selection and property editing.

Putting Paint on Paper

Now we can finally put paint on our bitmaps. This is all done by handlers in the `ChildForm` class.

This class performs two basic operations. The first is to place a shaped blob of color wherever the mouse is when the mouse button is pressed. The other is to allow the user to place a geometric shape and size it by dragging.

Listing 3.6.10 shows the full source of the `ChildForm` class interspersed with analysis.

LISTING 3.6.10 The `ChildForm` Class

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
```

LISTING 3.6.10 Continued

```
5: using System.ComponentModel;
6: using System.Windows.Forms;
7:
8: namespace FormPaint
9: {
10:    /// <summary>
11:    /// Summary description for ChildForm.
12:    /// </summary>
13:    public class ChildForm : System.Windows.Forms.Form
14:    {
15:        private Image myImage;
16:        private Bitmap tempBM;
17:
18:        private Pen pen;
19:        private SolidBrush brush;
20:        private GraphicsPath path;
21:        private Point firstPoint;
22:        private Point lastPoint;
23:        private Size blitBounds;
24:        private Point blitPos;
25:
26:        private bool Drawing;
27:        private bool dirty;
28:        private bool fill;
29:        private bool stroke;
30:
31:        private Graphics myGraphics;
32:        private Graphics imageGraphics;
33:
34:
35:        public Image Image
36:        {
37:            set
38:            {
39:                myImage = value;
40:                this.AutoScrollMinSize = myImage.Size;
41:                tempBM = new Bitmap(myImage.Size.Width, myImage.Size.Height);
42:            }
43:            get { return myImage; }
44:        }
```

The simple declaration of variables on lines 15–32 is followed by the `Image` accessor function that assigns an image to the child form and resets the `AutoScroll` limits to fit the picture. This accessor also allows other methods to get the image contained in the form.

LISTING 3.6.10 Continued

```
45:  
46:     public bool Dirty  
47:     {  
48:         get{return dirty;}  
49:         set{dirty=value;}  
50:     }  
51:  
52:     /// <summary>  
53:     /// Required designer variable.  
54:     /// </summary>  
55:     private System.ComponentModel.Container components = null;  
56:  
57:  
58:     public ChildForm()  
59:     {  
60:         //  
61:         // Required for Windows Form Designer support  
62:         //  
63:         InitializeComponent();  
64:  
65:         //  
66:         // TODO: Add any constructor code after InitializeComponent call  
67:         //  
68:         Drawing = false;  
69:         Dirty=false;  
70:         Text="Untitled.bmp";  
71:     }  
72:
```

The constructor on lines 58–71 calls the all-important `InitializeComponent`, sets up the initial values for the flags used to determine if the drawing has been changed or if a drawing action is taking place, and sets the text of the form to `untitled.bmp` in case this is a new image—not one loaded from disk.

```
73:     /// <summary>  
74:     /// Clean up any resources being used.  
75:     /// </summary>  
76:     protected override void Dispose( bool disposing )  
77:     {  
78:         if( disposing )  
79:         {  
80:             tempBM.Dispose();  
81:             if(components != null)  
82:             {  
83:                 components.Dispose();
```

LISTING 3.6.10 Continued

```
84:         }
85:     }
86:     base.Dispose( disposing );
87: }
88:
89: #region Windows Form Designer generated code
90: /// <summary>
91: /// Required method for Designer support - do not modify
92: /// the contents of this method with the code editor.
93: /// </summary>
94: private void InitializeComponent()
95: {
96:     //
97:     // ChildForm
98:     //
99:     this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
100:    this.BackColor = System.Drawing.Color.Green;
101:    this.ClientSize = new System.Drawing.Size(808, 565);
102:    this.Name = "ChildForm";
103:    this.Text = "ChildForm";
104:    this.MouseDown +=
105:        new System.Windows.Forms.MouseEventHandler(this.OnMouseDown);
106:    this.Closing +=
107:        new System.ComponentModel.CancelEventHandler(this.ChildForm_Closing);
108:    this.MouseUp +=
109:        new System.Windows.Forms.MouseEventHandler(this.OnMouseUp);
110:    this.Paint +=
111:        new System.Windows.Forms.PaintEventHandler(this.ChildForm_Paint);
```

```
112:    this.MouseMove +=
113:        new System.Windows.Forms.MouseEventHandler(this.OnMouseMove);
```

```
114: }
```

```
115: #endregion
```

The `Dispose` method and the `InitializeComponent` are added by the IDE. The only addition to `Dispose` is to clean up the intermediate bitmap. The handlers are added to the `ChildForm` on lines 104–108. On lines 113–116 following the `ToRadians` routine, simply convert from degrees to radians for use by the math routines.

```
116: }
117: 
```

LISTING 3.6.10 Continued

```
118:     private void CreateBrushPath(Shape s)
119:     {
120:
121:         path=new GraphicsPath();
122:
123:         MainForm form = (MainForm)this.ParentForm;
124:         int Toolsize=3;
125:         switch(form.CurrentTool)
126:         {
127:             case Tool.Paintbrush:
128:                 Toolsize = form.PaintBrushProperties.Size;
129:                 break;
130:             case Tool.Eraser:
131:                 Toolsize = form.EraserProperties.Size;
132:                 break;
133:         }
134:
135:         if(Toolsize<3)
136:             Toolsize=3;
137:         if(Toolsize>100)
138:             Toolsize=100;
139:
140:         switch(s)
141:         {
142:             case Shape.Round:
143:                 path.AddEllipse(-Toolsize/2,-Toolsize/2,
144:                                 Toolsize,Toolsize);
145:                 break;
146:             case Shape.Square:
147:                 path.AddRectangle(new Rectangle(-Toolsize/2,-Toolsize/2,
148:                                              Toolsize,Toolsize));
149:                 break;
150:             case Shape.Triangle:
151:                 Point[] points=new Point[3];
152:                 points[0]=new Point((int)(Math.Cos(ToRadians(90))*Toolsize),
153:                                 (int)(Math.Sin(ToRadians(90))*Toolsize));
154:                 points[1]=new Point((int)(Math.Cos(ToRadians(210))*Toolsize),
155:                                 (int)(Math.Sin(ToRadians(210))*Toolsize));
156:                 points[2]=new Point((int)(Math.Cos(ToRadians(330))*Toolsize),
157:                                 (int)(Math.Sin(ToRadians(330))*Toolsize));
158:                 path.AddPolygon(points);
159:                 break;
160:         }
```

LISTING 3.6.10 Continued

```
161:      }
162:
```

The `CreateBrushPath` method generates a path object that is used to fill a shaped area with color. In this simple demonstration, the color can be applied as a square, round, or triangular shape. The switch statement and its cases on lines 140–161 manage the creation of this path that is stored for use every time the mouse moves when the mouse button is pressed. Following that, on lines 163–171, the very simple paint routine does nothing more than copy the background image to the main screen.

```
163:      private void ChildForm_Paint(object sender,
164:                                     System.Windows.Forms.PaintEventArgs e)
165:      {
166:          e.Graphics.DrawImage(myImage,
167:                               this.AutoScrollPosition.X,
168:                               this.AutoScrollPosition.Y,
169:                               myImage.Width,
170:                               myImage.Height);
171:      }
172:
173:      private void OnMouseDown(object sender,
174:                               System.Windows.Forms.MouseEventArgs e)
175:      {
176:          this.Dirty=true;
177:
178:          if(this.MdiParent.ActiveMdiChild==this)
179:          {
180:              Color brushColor;
181:              myGraphics = this.CreateGraphics();
182:              imageGraphics = Graphics.FromImage(this.Image);
183:
184:              myGraphics.SetClip(new Rectangle(0,0,this.Image.Width,
185:                                              this.Image.Height));
186:
187:              MainForm form = (MainForm)this.MdiParent;
188:              switch(form.CurrentTool)
189:              {
190:                  case Tool.Paintbrush:
191:                      brushColor=Color.FromArgb(
192:                          255-(int)(255.0/
193:                                 100*form.PaintBrushProperties.Transparency),
194:                                 form.PaintBrushProperties.Color);
195:                      brush = new SolidBrush(brushColor);
196:                      CreateBrushPath(form.PaintBrushProperties.Shape);
```

LISTING 3.6.10 Continued

```
197:         break;
198:     case Tool.Eraser:
199:         brush = new SolidBrush(Color.White);
200:         CreateBrushPath(form.EraserProperties.Shape);
201:         break;
202:     case Tool.Rectangle:
203:         goto case Tool.Ellipse;
204:     case Tool.Ellipse:
205:         fill = form.ShapeProperties.Fill;
206:         stroke = form.ShapeProperties.Line;
207:         firstPoint=new Point(e.X,e.Y);
208:         lastPoint = firstPoint;
209:         pen = new Pen(form.ShapeProperties.LineColor,
210:             form.ShapeProperties.LineWidth);
211:         brush = new SolidBrush(form.ShapeProperties.FillColor);
212:         break;
213:     }
214: }
215:
216:     Drawing = true;
217:     OnMouseMove(sender,e);
218:
219: }
```

The `OnMouseDown` handler is the first action in the drawing process. It handles two main cases. If the mouse button is pressed and the paint or eraser tool is selected, it makes a new paintbrush path of the correct size and shape then sets the brush colors and transparency. In the case of a shape tool, it stores the first point in the shape, usually the top-left corner point, and then defines the pen color and width followed by the brush color. Finally, this method calls the mouse move handler once to ensure that the first paint is applied to the paper.

Following, on lines 221–257, the mouse up handler only deals with the final painting of the desired shape—either a rectangle or ellipse—on the canvas. The paintbrush or eraser color is applied on every mouse move. This drawing is performed not on the screen but on the image being held in memory by the form. This image is never scrolled, so the positions of the mouse and scrollbars must be compensated for on lines 229–232.

```
220:
221:     private void OnMouseUp(object sender,
222:         System.Windows.Forms.MouseEventArgs e)
223:     {
224:
225:         if(this.MdiParent.ActiveMdiChild==this)
```

LISTING 3.6.10 Continued

```
226:      {
227:          Point topLeft;
228:          Size bounds;
229:          topLeft=new Point(Math.Min(this.firstPoint.X,e.X),
230:              Math.Min(this.firstPoint.Y,e.Y));
231:          bounds=new Size(Math.Abs(e.X-firstPoint.X),
232:              Math.Abs(e.Y-firstPoint.Y));
233:          topLeft.Offset(-AutoScrollPosition.X,-AutoScrollPosition.Y);
234:          MainForm form = (MainForm)this.MdiParent;
235:          switch(form.CurrentTool)
236:          {
237:              case Tool.Rectangle:
238:                  if(fill)
239:                      imageGraphics.FillRectangle(brush,topLeft.X,
240:                          topLeft.Y,bounds.Width,bounds.Height);
241:                  if(stroke)
242:                      imageGraphics.DrawRectangle(pen,topLeft.X,
243:                          topLeft.Y,bounds.Width,bounds.Height);
244:                  break;
245:              case Tool.Ellipse:
246:                  if(fill)
247:                      imageGraphics.FillEllipse(brush,topLeft.X,
248:                          topLeft.Y,bounds.Width,bounds.Height);
249:                  if(stroke)
250:                      imageGraphics.DrawEllipse(pen,topLeft.X,
251:                          topLeft.Y,bounds.Width,bounds.Height);
252:                  break;
253:          }
254:      }
255:      Drawing=false;
256:      Invalidate();
257:  }
258:
259:  private void OnMouseMove(object sender,
260:      System.Windows.Forms.MouseEventArgs e)
261:  {
262:      if(!Drawing)
263:          return;
264:      if(this.MdiParent.ActiveMdiChild==this)
265:      {
266:          Graphics gTemp=Graphics.FromImage(tempBM);
267:          MainForm form = (MainForm)this.MdiParent;
268:          if(form.CurrentTool==Tool.Ellipse ||
269:              form.CurrentTool==Tool.Rectangle)
```

LISTING 3.6.10 Continued

```
270:      {
271:          blitPos= new Point(
272:              Math.Min(Math.Min(e.X,firstPoint.X),lastPoint.X),
273:              Math.Min(Math.Min(e.Y,firstPoint.Y),lastPoint.Y));
274:          blitBounds = new Size(Math.Max(e.X,
275:              Math.Max(firstPoint.X,lastPoint.X))-blitPos.X,
276:              Math.Max(e.Y,Math.Max(firstPoint.Y,lastPoint.Y))-
277:              blitPos.Y);
278:          blitPos.Offset(-(int)(1+pen.Width/2),-(int)(1+pen.Width/2));
279:          blitBounds.Width+=(int)(2+pen.Width);
280:          blitBounds.Height+=(int)(2+pen.Width);
281:          form.StatusText=blitPos.ToString()+" "+blitBounds.ToString();
282:      }
283:      switch(form.CurrentTool)
284:      {
285:          case Tool.Paintbrush:
286:              GraphicsContainer ctr=myGraphics.BeginContainer();
287:              myGraphics.Transform.Reset();
288:              myGraphics.TranslateTransform(e.X,e.Y);
289:              myGraphics.FillPath(this.brush,this.path);
290:              myGraphics.EndContainer(ctr);
291:              ctr=imageGraphics.BeginContainer();
292:              imageGraphics.Transform.Reset();
293:              imageGraphics.TranslateTransform(e.X-AutoScrollPosition.X,
294:                  e.Y-AutoScrollPosition.Y);
295:              imageGraphics.FillPath(this.brush,this.path);
296:              imageGraphics.EndContainer(ctr);
297:
298:          break;
299:          case Tool.Eraser:
300:              goto case Tool.Paintbrush;
301:          case Tool.Rectangle:
302:              gTemp.DrawImage(myImage,
303:                  new Rectangle(blitPos,blitBounds),
304:                  blitPos.X-AutoScrollPosition.X,
305:                  blitPos.Y-AutoScrollPosition.Y,
306:                  blitBounds.Width,blitBounds.Height,
307:                  GraphicsUnit.Pixel);
308:              if(fill)
309:                  gTemp.FillRectangle(brush,
310:                      Math.Min(this.firstPoint.X,e.X),
311:                      Math.Min(this.firstPoint.Y,e.Y),
312:                      Math.Abs(e.X-firstPoint.X),
313:                      Math.Abs(e.Y-firstPoint.Y));
```

LISTING 3.6.10 Continued

```
314:         if(stroke)
315:             gTemp.DrawRectangle(pen,
316:                 Math.Min(this.firstPoint.X,e.X),
317:                 Math.Min(this.firstPoint.Y,e.Y),
318:                 Math.Abs(e.X-firstPoint.X),
319:                 Math.Abs(e.Y-firstPoint.Y));
320:             myGraphics.DrawImage(tempBM,
321:                 new Rectangle(blitPos,blitBounds),
322:                 blitPos.X,blitPos.Y,
323:                 blitBounds.Width,
324:                 blitBounds.Height,
325:                 GraphicsUnit.Pixel);
326:         break;
327:     case Tool.Ellipse:
328:         gTemp.DrawImage(myImage,
329:             new Rectangle(blitPos,blitBounds),
330:             blitPos.X-AutoScrollPosition.X,
331:             blitPos.Y-AutoScrollPosition.Y,
332:             blitBounds.Width,
333:             blitBounds.Height,
334:             GraphicsUnit.Pixel);
335:         if(fill)
336:             gTemp.FillEllipse(brush,
337:                 Math.Min(this.firstPoint.X,e.X),
338:                 Math.Min(this.firstPoint.Y,e.Y),
339:                 Math.Abs(e.X-firstPoint.X),
340:                 Math.Abs(e.Y-firstPoint.Y));
341:         if(stroke)
342:             gTemp.DrawEllipse(pen,
343:                 Math.Min(this.firstPoint.X,e.X),
344:                 Math.Min(this.firstPoint.Y,e.Y),
345:                 Math.Abs(e.X-firstPoint.X),
346:                 Math.Abs(e.Y-firstPoint.Y));
347:             myGraphics.DrawImage(tempBM,
348:                 new Rectangle(blitPos,blitBounds),
349:                 blitPos.X,blitPos.Y,
350:                 blitBounds.Width,
351:                 blitBounds.Height,
352:                 GraphicsUnit.Pixel);
353:         break;
354:     }
355:     lastPoint.X=e.X;
356:     lastPoint.Y=e.Y;
```

LISTING 3.6.10 Continued

```
357:      }
358: }
```

The `OnMouseMove` method on lines 259–358 is the work-horse routine in the program. It works in two modes, differentiating between paintbrush or eraser operations and shape drawing operations. In the former mode, the paint is applied to the bitmap using the `GraphicsPath` created in the `OnMouseDown` event. The color is placed in both the screen memory and on the internal image. This means that a paint operation is instant and affects the image immediately. This takes place on lines 285–300 with line 289 performing the paint onscreen and line 295 placing color on the background image.

The operations that take place in shape drawing are more complex. As a shape is drawn, the user might want to “rubberband” the shape around to get the size correct. This means that paint cannot be placed directly onto the background image. A double buffering technique is used where an intermediate image is used to composite an area from the background with the graphical shape. This image is then copied to the screen, and the process repeated until the mouse button is released. This technique removes any flicker caused by refreshing and repainting the shape directly on the screen.

Line 269 decides if a `Rectangle` or `Ellipse` operation is underway. If so, the maximum size of the affected screen area, taking into account the start position, the current position, and the previous position of the mouse on lines 271 and 274. The area is increased by half a line width all around to compensate for the thickness of the current pen on lines 278–280 and then the double-buffered draw for rectangles happens on lines 301–326 and for ellipses on lines 327–353.

Also note line 281, which updates the status bar with the shape position and size.

```
359:
360:     private void ChildForm_Closing(object sender,
361:         System.ComponentModel.CancelEventArgs e)
362:     {
363:         if(Dirty)
364:         {
365:             DialogResult result=MessageBox.Show(this,
366:                 "This file has changed, do you wish to save",
367:                 "Save file",
368:                 MessageBoxButtons.YesNoCancel,
369:                 MessageBoxIcon.Question);
370:             switch(result)
371:             {
372:                 case DialogResult.Cancel:
373:                     e.Cancel=true;
374:                     break;
```

LISTING 3.6.10 Continued

```
375:         case DialogResult.No:
376:             break;
377:         case DialogResult.Yes:
378:             this.Image.Save(this.Text);
379:             break;
380:         }
381:     }
382: }
383:
384: protected override void OnPaintBackground(PaintEventArgs e)
385: {
386:     Region r=new Region(new Rectangle(0,0,Image.Width,Image.Height));
387:     Region w=new Region(new Rectangle(AutoScrollPosition.X,
388:         AutoScrollPosition.Y,
389:         ClientRectangle.Width,
390:         ClientRectangle.Height));
391:     r.Complement(w);
392:     e.Graphics.FillRegion(new SolidBrush(this.BackColor),r);
393: }
394:
395: protected override void OnSizeChanged(EventArgs e)
396: {
397:     Invalidate();
398:     base.OnSizeChanged(e);
399: }
400: }
401: }
```

Finally, the last few methods deal with saving the file if it has been altered (lines 360–381). This handler is called before the form closes to see if it is allowed to continue. The message box used on lines 365–369 determines if the user wants to save or discard the image or cancel the close altogether. In the case of a cancel, the `CancelEventArgs` provided through the `delegate` call must be updated. This happens on line 373.

The `OnPaintBackground` override is provided to eliminate flicker. With a green background and the default settings, the whole bitmap is over-painted with green before the image is redrawn. This caused a nasty flicker unless the steps taken here are used. This method calculates the region of screen outside of the image by using the complement of this area and the area covered by the image (lines 386–391) and only repaints the piece required.

The `OnSizeChanged` override of the base class method in lines 395–399 ensures that the background is correctly repainted by just invalidating the window.

The image in Figure 3.6.10 shows `FormPaint` working, albeit with some abysmal artistic talent.

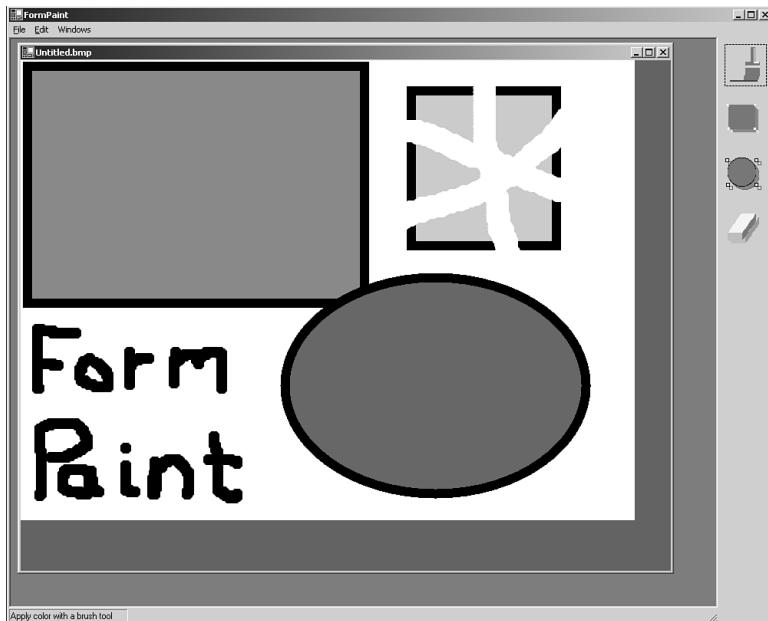


FIGURE 3.6.10

The working `FormPaint` application.

3.6

PRACTICAL
WINDOWS FORMS
APPLICATIONS

Summary

In this section's final chapter, you've seen the property grid, how to create a working Windows Forms application with a step-by-step walkthrough, and how to enhance the user experience by customizing controls.

We hope that this part of the book will enable you to get a running start at the Windows Forms development system, and that you begin to produce powerful and productive applications.

In the next section of this book, "Web Technologies," we'll be examining the details of assemblies and how to sign and version your code to put a stop to DLL hell once and for all.

WEEK 2

DAY 13

Introducing Web Services

Today's lesson covers the fundamentals of creating and using Web Services, an exciting new way of creating programs that anyone with access to the Internet can use. Although Web Services use sophisticated mechanisms and protocols to communicate with clients, .NET makes the job of creating and using Web Services easy.

NEW TERM

Web Services are programs with methods that can be called over the Internet. Web Services allow you to create *distributed applications*, or applications that live on a number of different computers. The .NET framework and ASP.NET provide all the underlying mechanisms to make your program Web accessible. .NET provides all the plumbing for Web Services so that you can focus on developing your application.

Today you will learn the following:

- Why Web Services are a viable technology
- How to create a Web Service
- How to create Web Service clients with the .NET framework utilities
- How to create Web Service clients with Visual Studio.NET
- How Web Services work

Why Use Web Services?

Already, a number of technologies allow you to create distributed applications, including CORBA, DCOM, RPC, and custom solutions based on TCP sockets. Because these technologies are already in place and are quite mature at this time, why would you or your company make the leap to Web Services?

You might choose Web Services for a number of good reasons if you plan to make a distributed application:

- Web Services are built on industry standard protocols, such as TCP, SSL, XML, SOAP, and WSDL. We'll explain these protocols in detail later today.
- Web Services are easy—even trivial—to implement with .NET. If you've ever tried to implement a CORBA or DCOM solution, you'll be pleasantly surprised by how painless Web Service implementation and configuration are with .NET and ASP.NET.
- You don't need a lot of infrastructure in place to develop a Web Service. The only real requirements are a Web server and a connection to the Internet.

Implementing Your First Web Service

Rather than dwell on the theoretical aspects of Web Services, let's implement a simple Web Service to demonstrate some of the points from the preceding section. Two pieces are required in any Web Service implementation: the service and the client. We will create the service first.

The easiest way to create Web Services in .NET is to build them using ASP.NET. Web Service files in ASP.NET have the special filename extension .asmx. ASP.NET flags any file with this extension as a Web Service and compiles it as a Web Service.

The first sample service that we will create simply returns the current time on the server as a string. Follow these steps to create the service:

1. As explained for the first ASP.NET page on Day 2, “Introducing ASP.NET,” create a directory for the Web Service project on your computer. For this example, use the directory C:\src\timeservice.
2. On your Web server, create a virtual directory named TimeService that points to the project directory in step 1. (Day 2 explained how to create virtual directories.)
3. In the project directory, create a new file called TimeUtilities.asmx using your text editor or download this file from the book’s Web site at www.samspublishing.com. Type the contents of Listing 13.1 if you didn’t download the file.

Listing 13.1 shows a basic Web Service that will return the time of day as a string on the server.

LISTING 13.1 TimeUtils.asmx: Returning the Current Time on the Server

```
1: <%@ WebService Language="C#" Class="TimeUtilities" %>
2: using System;
3: using System.Web.Services;
4:
5: [WebService(Namespace="http://tempuri.org/webservices")]
6: public class TimeUtilities : WebService
7: {
8:     [WebMethod]
9:     public String GetTime()
10:    {
11:        String ret = "The current time on the server is: ";
12:        String curTime = DateTime.Now.ToString();
13:        return ret + curTime;
14:    }
15: }
```

ANALYSIS Line 1 uses the `WebService` page directive. You've seen other ASP.NET page directives such as `Page`, `Import`, and `Control` on previous days. The `WebService` page directive marks the remaining code in the page as a Web Service. Within the page directive, you must specify the class of the Web Service that you are going to define in the code. Each Web Service must have its own class; the one in this example is called `TimeUtilities`.

Lines 2 and 3 specify that we are using the `System` and `System.Web.Services` namespaces. They are required in any Web Service that we write.

On Line 5 comes the definition of the class. Notice that before we define the class, we use an attribute for it:

```
[WebService(Namespace="http://tempuri.org/webservices")]
```

NEW TERM This line is called an *attribute definition*. If you are an experienced .NET developer, you may already have used these kinds of attributes in your code. If you haven't used them, just understand that they supplement the code you write. In this case, we are using the `WebService` attribute to specify the namespace for our Web Service.

Because Web Services can be exposed to the rest of the world on the Internet, each one needs a unique namespace. This namespace is exactly the same kind of thing you would use in an XML file that you were going to share with another company, as we explained on Day 8, "Using XML with Your Application." If you are developing a Web Service for an organization, you should use your organization's public URL instead of `tempuri.org`.

Next, Line 6 defines the `TimeUtilities` class. Because the class defines a Web Service, it must derive from the `WebService` class in the `System.Web.WebServices` namespace.

Our next order of business (Lines 9–14) is to define the method that we want our Web Service users to call. Notice that the `GetTime` method is preceded by the `[WebMethod]` attribute. All methods that should be available to Web Service users must have this attribute. `GetTime` simply returns a string with the current time on the server.

Testing the Web Service

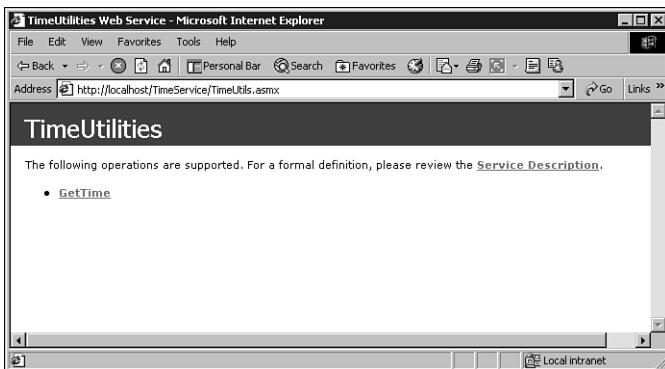
To test the Web Service, you need to point a Web browser to the Web Service as if it were a regular Web page. ASP.NET automatically generates some simple documentation for the Web Service and provides Web pages to invoke methods on the service. If you're using the same computer where the Web Service resides to test the service, point your browser to the following URL. If not, substitute the name of the computer for `localhost` in the following URL:

`http://localhost/TimeService/TimeUtils.asmx`

You should see a page that resembles Figure 13.1. From the figure, you can see that ASP.NET has created a page with a list of the methods that the `TimeUtils` Web Service exposes.

FIGURE 13.1

ASP.NET output for the TimeUtils.



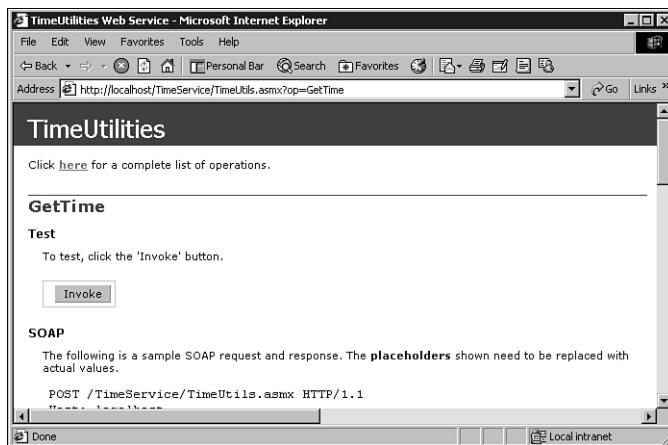
Click the `GetTime` link to see a page that has some template documentation for the `GetTime` method, as shown in Figure 13.2.

By clicking the Invoke button on this page, you see a response similar to the following code lines in your browser. The Web server (IIS) returns an XML document that contains the response from the Web Service.

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/webservices">
    The current time on the server is: 7/7/2001 3:55:41 PM
</string>
```

FIGURE 13.2

ASP.NET boilerplate documentation for the GetTime method.



Describing the Web Service

Web Services need a way to tell client programs what methods they implement, the parameters each method takes, and the return values each method gives back. They do so through a WSDL file. WSDL, which stands for Web Services Description Language, provides clients of a Web Service with all the necessary details they need to connect to and use the Web Service.

WSDL files are XML files. To see what the WSDL file looks like for our TimeUtils application, point your browser to the URL of the TimeUtils.asmx file and append the string ?WSDL to the URL, as in the following line:

`http://localhost/TimeService/TimeUtils.asmx?WSDL`

Tip

You can also see the WSDL file for the TimeUtils application by browsing to the main Web Service page and then clicking the Service Description link (refer to Figure 13.1).

13

The WSDL file returned from our sample Web Service looks like Listing 13.2. The WSDL is quite large, even for our simple Web Service. Luckily, you should never have to create WSDL files yourself; the Web Service will generate the files automatically using the plumbing built into ASP.NET.

LISTING 13.2 The WSDL File for the TimeUtils Web Service

```
1:  <?xml version="1.0" encoding="utf-8" ?>
2:  <definitions
3:      xmlns:s="http://www.w3.org/2001/XMLSchema"
4:      xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
5:      xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
6:      xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
7:      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8:      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
9:      xmlns:s0="http://tempuri.org/webservices"
10:     targetNamespace="http://tempuri.org/webservices"
11:     xmlns="http://schemas.xmlsoap.org/wsdl/">
12:  <types>
13:      <s:schema attributeFormDefault="qualified"
14:          elementFormDefault="qualified"
15:          targetNamespace="http://tempuri.org/webservices">
16:          <s:element name="GetTime">
17:              <s:complexType />
18:          </s:element>
19:          <s:element name="GetTimeResponse">
20:              <s:complexType>
21:                  <s:sequence>
22:                      <s:element
23:                          minOccurs="1"
24:                          maxOccurs="1"
25:                          name="GetTimeResult"
26:                          nillable="true"
27:                          type="s:string" />
28:                  </s:sequence>
29:              </s:complexType>
30:          </s:element>
31:          <s:element name="string" nillable="true" type="s:string" />
32:      </s:schema>
33:  </types>
34:  <message name="GetTimeSoapIn">
35:      <part name="parameters" element="s0:GetTime" />
36:  </message>
37:  <message name="GetTimeSoapOut">
38:      <part name="parameters" element="s0:GetTimeResponse" />
39:  </message>
40:  <message name="GetTimeHttpGetIn" />
41:  <message name="GetTimeHttpGetGetOut">
42:      <part name="Body" element="s0:string" />
43:  </message>
44:  <message name="GetTimeHttpPostIn" />
45:  <message name="GetTimeHttpPostOut">
46:      <part name="Body" element="s0:string" />
47:  </message>
48:  <portType name="TimeUtilitiesSoap">
```

LISTING 13.2 Continued

```
49:      <operation name="GetTime">
50:        <input message="s0:GetTimeSoapIn" />
51:        <output message="s0:GetTimeSoapOut" />
52:      </operation>
53:    </portType>
54:  <portType name="TimeUtilitiesHttpGet">
55:    <operation name="GetTime">
56:      <input message="s0:GetTimeHttpGetIn" />
57:      <output message="s0:GetTimeHttpGetOut" />
58:    </operation>
59:  </portType>
60:  <portType name="TimeUtilitiesHttpPost">
61:    <operation name="GetTime">
62:      <input message="s0:GetTimeHttpPostIn" />
63:      <output message="s0:GetTimeHttpPostOut" />
64:    </operation>
65:  </portType>
66:  <binding name="TimeUtilitiesSoap" type="s0:TimeUtilitiesSoap">
67:    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
68:                  style="document" />
69:    <operation name="GetTime">
70:      <soap:operation soapAction="http://tempuri.org/webservices/GetTime"
71:                      style="document" />
72:      <input>
73:        <soap:body use="literal" />
74:      </input>
75:      <output>
76:        <soap:body use="literal" />
77:      </output>
78:    </operation>
79:  </binding>
80:  <binding name="TimeUtilitiesHttpGet" type="s0:TimeUtilitiesHttpGet">
81:    <http:binding verb="GET" />
82:    <operation name="GetTime">
83:      <http:operation location="/GetTime" />
84:      <input>
85:        <http:urlEncoded />
86:      </input>
87:      <output>
88:        <mime:mimeXml part="Body" />
89:      </output>
90:    </operation>
91:  </binding>
92:  <binding name="TimeUtilitiesHttpPost" type="s0:TimeUtilitiesHttpPost">
93:    <http:binding verb="POST" />
94:    <operation name="GetTime">
95:      <http:operation location="/GetTime" />
96:      <input>
97:        <mime:content type="application/x-www-form-urlencoded" />
```

LISTING 13.2 Continued

```
98:      </input>
99:      <output>
100:         <mime:mimeXml part="Body" />
101:      </output>
102:     </operation>
103:   </binding>
104:   <service name="TimeUtilities">
105:     <port name="TimeUtilitiesSoap" binding="s0:TimeUtilitiesSoap">
106:       <soap:address
107:          location="http://localhost/timeservice/timeutils.asmx" />
108:        </port>
109:     <port name="TimeUtilitiesHttpGet" binding="s0:TimeUtilitiesHttpGet">
110:       <http:address
111:          location="http://localhost/timeservice/timeutils.asmx" />
112:        </port>
113:     <port name="TimeUtilitiesHttpPost"
114:           binding="s0:TimeUtilitiesHttpPost">
115:       <http:address
116:          location="http://localhost/timeservice/timeutils.asmx" />
117:        </port>
118:     </service>
119:   </definitions>
```

Analyzing the WSDL File

ANALYSIS To get a general sense of what the WSDL file describes, let's take a detailed look at Listing 13.2. The top-level XML nodes for the TimeUtils WSDL file are as follows:

```
<definitions>
  <types>

    <message name="GetTimeSoapIn">
    <message name="GetTimeSoapOut">
    <message name="GetTimeHttpGetIn">
    <message name="GetTimeHttpGetOut">
    <message name="GetTimeHttpPostIn">
    <message name="GetTimeHttpPostOut">

    <portType name="TimeUtilitiesSoap">
    <portType name="TimeUtilitiesHttpGet">
    <portType name="TimeUtilitiesHttpPost">

    <binding name="TimeUtilitiesSoap">
    <binding name="TimeUtilitiesHttpGet">
    <binding name="TimeUtilitiesHttpPost">

    <service name="TimeUtilities">
  </definitions>
```

You can see that the WSDL file contains several kinds of nodes: `types`, `message`, `portType`, `binding`, and `service`. The `types` node describes the names, parameters, and return values of each method in the Web Service. The `message`, `portType`, `binding`, and `service` definitions describe how a client should communicate with the Web Service.

Let's analyze each WSDL definition node in reverse order.

The service Node

The `service` node defines the name of our Web Service—in this case, `TimeUtilities`—and then defines ports:

```
104:    <service name="TimeUtilities">
105:      <port name="TimeUtilitiesSoap" binding="s0:TimeUtilitiesSoap">
106:        <soap:address
107:          location="http://localhost/timeservice/timeutils.asmx" />
108:        </port>
109:        <port name="TimeUtilitiesHttpGet" binding="s0:TimeUtilitiesHttpGet">
110:          <http:address
111:            location="http://localhost/timeservice/timeutils.asmx" />
112:          </port>
113:          <port name="TimeUtilitiesHttpPost"
114:            binding="s0:TimeUtilitiesHttpPost">
115:            <http:address
116:              location="http://localhost/timeservice/timeutils.asmx" />
117:            </port>
118:          </service>
```

Each port corresponds to a certain protocol that the Web Service will support. Our Web Service supports the SOAP, HTTP Post, and HTTP Get protocols.



Note

Not every Web Service supports HTTP, because some complex parameters can't be passed in and out of Web Services using HTTP. We'll explain the rules surrounding when HTTP is supported later today. SOAP supports any kind of data type.

13

The portType Node

Each port supports an “operation.” Each operation corresponds to a method that the Web Service supports. You can see that the following SOAP `portType` supports the `GetTime` operation for our Web Service. Each operation consists of “messages”: an input message for when the method is called and an output message for when the method returns a value.

```
48:   <portType name="TimeUtilitiesSoap">
49:     <operation name="GetTime">
50:       <input message="s0:GetTimeSoapIn" />
51:       <output message="s0:GetTimeSoapOut" />
52:     </operation>
53:   </portType>
```

The message Node

The message definition for the `GetTimeSoapIn` method is as follows:

```
34:   <message name="GetTimeSoapIn">
35:     <part name="parameters" element="s0:GetTime" />
36:   </message>
```

The message definition describes how parameters will be passed into and out of the Web Service. The `GetTime` method doesn't receive any parameters, so the message definition is trivial.

The types and schema Nodes

The types node describes each method, parameter, and return value for a Web Service. These Web Service elements are detailed in a schema node. A simplified version of the schema node for our Web Service follows:

```
<s:schema>
  <s:element name="GetTime">
    <s:complexType />
  </s:element>
  <s:element name="GetTimeResponse">
    <s:complexType>
      <s:sequence>
        <s:element name="GetTimeResult" type="s:string" />
      </s:sequence>
    </s:complexType>
  </s:element>
  <s:element name="string" nillable="true" type="s:string" />
</s:schema>
```

Here, the schema node describes the `GetTime` and `GetTimeResponse` messages that our Web Service deals with. Our Web Service returns a `GetTimeResponse` message after the `GetTime` method is invoked. The definition shows that the response message includes a string.

Implementing the Web Service Client

Although being able to access a Web Service from a Web browser is nice, a more realistic use is to create custom client programs that access a Web Service remotely, such as Web pages and standalone applications. To demonstrate how to create such programs, we'll create two different clients for our TimeUtils Web Service: a console application and a Web page.

Tip

You can create Web Service clients easily by using Visual Studio.NET; we'll explain how to create them a little later today. However, to show some of the "plumbing" behind Web Services, let's continue using a text editor to create our client programs.

NEW TERM

Before creating the client to our TimeUtils application, we need to create a proxy class. A *proxy class* allows us to call the `GetTime` method without worrying about how to connect to the server and how to pass and return parameters to and from the method. In general, a *proxy* is an object that wraps up method calls and parameters, forwards them to a remote host, and returns the results to us. Proxies serve the role as a middleman on the client computer.

.NET comes with a special utility called WSDL.exe for creating Web Service proxies. As you might imagine from the utility's name, WSDL.exe takes a Web Service's WSDL description and creates a proxy. Follow these steps to create the proxy with WSDL.exe:

1. Open a Visual Studio.NET command prompt (from the Start menu, choose Visual Studio.NET, Visual Studio.NET Tools, and then Visual Studio.Net Command Prompt). This DOS command prompt contains all the paths for the .NET tools.
2. Create a new directory for the client projects, such as C:\src\timeclient.
3. Type the following command at the prompt (try to fit the entire command at the second prompt onto one line):

```
C:>cd src\timeclient  
C:\src\timeclient> wsdl /l:CS /n:WebBook  
  ↵/out:TimeProxy.cs  
  ↵http://localhost/TimeService/TimeUtils.asmx?WSDL
```

This command creates the `TimeProxy.cs` file, which you can use in your client applications:

- `/l:CS` tells the utility to create the proxy using C#.
 - `/n:WebBook` specifies a namespace for the WSDL tools to use when it creates the proxy class. You can use any namespace name you want.
 - `/out:TimeProxy.cs` specifies the name of the output file.
 - The last parameter is an URL for getting a WSDL description of the Web Service. You can also specify a file containing WSDL. However, if you use an URL, as in this example, the utility will automatically browse to the Web address and use the WSDL file that it finds.
4. Compile the new proxy file:

```
C:\src\timeclient> csc /t:library TimeProxy.cs
```

The proxy that the WSDL.exe utility creates should look like the file shown in Listing 13.3.

LISTING 13.3 An Automatically Generated Proxy Class for the TimeUtils Client

```
1: //-----
2: // <autogenerated>
3: //   This code was generated by a tool.
4: //   Runtime Version: 1.0.2914.16
5: //
6: //   Changes to this file may cause incorrect behavior and will be lost
7: //   if the code is regenerated.
8: // </autogenerated>
9: //-----
10:
11: //
12: // This source code was auto-generated by wsdl, Version=1.0.2914.16.
13: //
14: namespace WebBook {
15:     using System.Diagnostics;
16:     using System.Xml.Serialization;
17:     using System;
18:     using System.Web.Services.Protocols;
19:     using System.Web.Services;
20:
21:     [System.Web.Services.WebServiceBindingAttribute(
22:         Name="TimeUtilitiesSoap",
23:         Namespace="http://tempuri.org/webservices")]
24:     public class TimeUtilities :
25:         System.Web.Services.Protocols.SoapHttpClientProtocol {
26:
27:         [System.Diagnostics.DebuggerStepThroughAttribute()]
28:         public TimeUtilities() {
29:             this.Url = "http://localhost/TimeService/TimeUtils.asmx";
30:         }
31:
32:         [System.Diagnostics.DebuggerStepThroughAttribute()]
33:         [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
34:             "http://tempuri.org/webservices/GetTime",
35:             RequestNamespace="http://tempuri.org/webservices",
36:             ResponseNamespace="http://tempuri.org/webservices",
37:             Use=System.Web.Services.Description.SoapBindingUse.Literal,
38:             ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
39:         public string GetTime() {
40:             object[] results = this.Invoke("GetTime", new object[0]);
41:             return ((string)(results[0]));
42:         }
43:
44:         [System.Diagnostics.DebuggerStepThroughAttribute()]
45:         public System.IAsyncResult
46:             BeginGetTime(System.AsyncCallback callback,
```

LISTING 13.3 Continued

```
47:                     object asyncState) {
48:             return this.BeginInvoke("GetTime", new object[0],
49:                                     callback, asyncState);
50:         }
51:
52:         [System.Diagnostics.DebuggerStepThrough()]
53:         public string EndGetTime(System.IAsyncResult asyncResult) {
54:             object[] results = this.EndInvoke(asyncResult);
55:             return ((string)(results[0]));
56:         }
57:     }
58: }
```

ANALYSIS The proxy file contains a new definition for the `TimeUtilities` class, different from the one created earlier in Listing 13.1. This new definition is the version that all our client programs will use. Although this new class is defined differently than the original created for the Web Service, this new `TimeUtilities` class will work the same way for any client program that uses it. Rather than execute code directly, this new class will call the Web Service to execute each method.

Notice also the two new methods in the proxy class, `BeginGetTime` (Lines 44–50) and `EndGetTime` (Lines 52–56). These methods are created so that we can call the Web Service asynchronously. This means that we can call the Web Service in our client program and then do other work immediately. Then, when the Web Service returns a result, we can process the results at that time. This asynchronous method for calling remote Web Services might not be responsive because of a slow network connection. Day 16, “Putting It All Together with Web Services,” will contain detailed samples that show how to use asynchronous calling methods and Web Services.

Now that we’ve created our proxy class, we can create client programs that use our Web Service. To begin, let’s create a simple console application to call the Web Service. The client is shown in Listing 13.4.

LISTING 13.4 CallService.cs: A Console Application That Calls the TimeUtils Web Service

```
using System;
using WebBook;

namespace WebBook
{
    public class CallService
    {
        public static void Main()
        {
```

LISTING 13.4 Continued

```
TimeUtilities remoteService = new TimeUtilities();

Console.WriteLine("Calling web service...");

Console.WriteLine(remoteService.GetTime());
}

}
```

If you save the CallService.cs client in the `TimeClient` project directory that you created, you can compile it by using this command:

```
csc CallService.cs /r:TimeProxy.dll
```

This command creates a file named `CallService.exe` in the client directory, which you can run now. For this command to work, you must have compiled the proxy class in Listing 13.3 according to the instructions in step 4 of this section.

After running the `CallService.exe` program you created, you should see output like the following:

OUTPUT Calling web service...
The current time on the server is: 7/9/2001 12:19:08 PM

Creating a Web Service Client

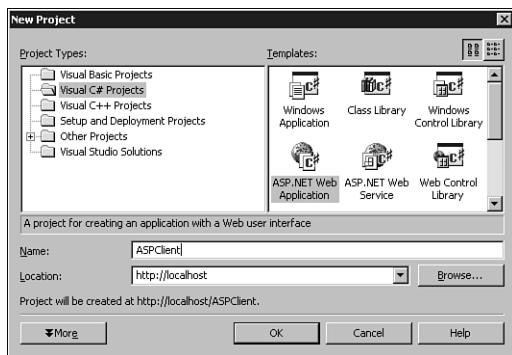
Now that you've seen how to create a Web Service proxy and client program manually, let's use Visual Studio.NET to make a Web page that calls the `TimeUtils` Web Service. Visual Studio.NET automates the process by creating a proxy file using a wizard. For the example in this section, we'll create an ASP.NET page that calls the Web Service.

To create the client application, follow these steps:

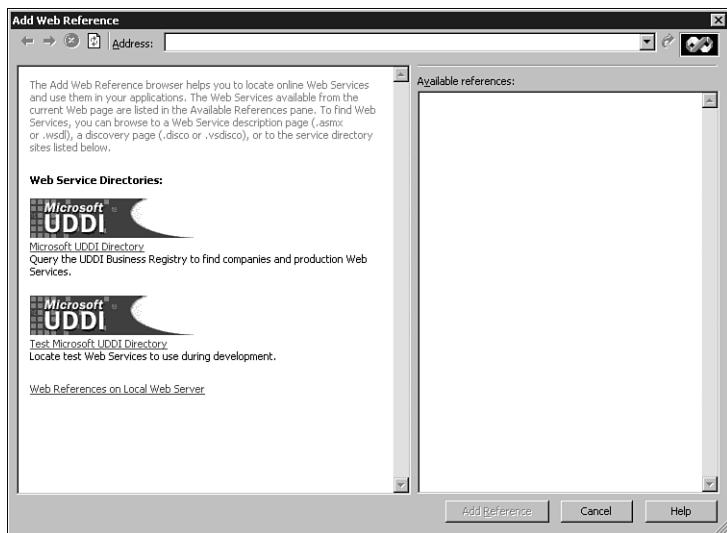
1. Create a new ASP.NET Web application by selecting New Project on Visual Studio's start page.
2. In the New Project dialog, select Visual C# Projects in the Project Types list and then select ASP.NET Web Application from the Templates list. Name the Web application `ASPClient` and click OK (see Figure 13.3).
3. From the Project menu, choose Add Web Reference. You should see a dialog similar to Figure 13.4.
4. In the Address text box, enter `http://localhost/TimeService/TimeUtils.asmx` and then click the Add Reference button to create a proxy file for the Web Service and add it to your project.

FIGURE 13.3

The New Project dialog.

**FIGURE 13.4**

The Add Web Reference dialog.



5. Change the `Page_Load` method in `WebForm1.aspx` to use the code in Listing 13.5.
6. Run the project.

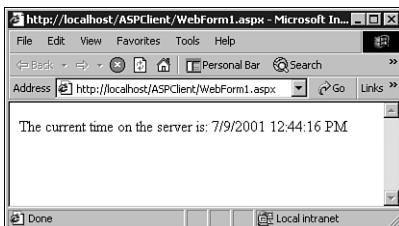
13**LISTING 13.5** Page_Load Method for the Visual Studio Client ASP.NET Page

```
private void Page_Load(object sender, System.EventArgs e)
{
    localhost.TimeUtilities remoteService = new localhost.TimeUtilities();
    Response.Write(remoteService.GetTime());
}
```

If the remote Web Service resides on another machine, the `localhost` string in Listing 13.5 changes to the name of that machine. After running the project, you should see a screen similar to Figure 13.5.

FIGURE 13.5

The ASP.NET client for the Web Service.

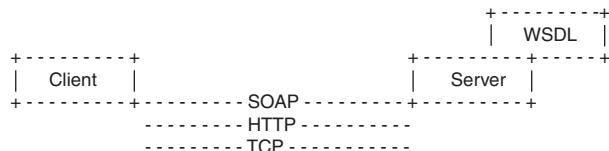


Understanding How Web Services Work

Web Services communicate with clients using SOAP (Simple Object Access Protocol), which is built on top of the HTTP and TCP Internet protocols. Figure 13.6 shows how SOAP is used with these other two protocols between a Web Service and its client.

FIGURE 13.6

Internet protocols used by a Web Service and its clients.



SOAP uses XML and defines how a function should be called on a remote machine. It also specifies how parameters should be passed to and from a function on a remote machine. For example, Listing 13.6 shows the SOAP message that a client sends to the TimeUtils Web Service.

LISTING 13.6 A SOAP Request to the TimeUtils Web Service

```
1: <?xml version="1.0" encoding="utf-8"?>
2: <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3:           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4:           xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
5:   <soap:Body>
6:     <GetTime xmlns="http://tempuri.org/webservices" />
7:   </soap:Body>
8: </soap:Envelope>
```

ANALYSIS From Listing 13.6, you can see that each SOAP request is an XML file that contains the name of a method to call and any parameters that the method might need. Because SOAP uses the HTTP protocol, the HTTP headers (Lines 1–5) in Listing 13.7 are placed before the SOAP message (Lines 2–8) in Listing 13.6. The HTTP request using SOAP for the TimeUtils looks like Listing 13.7.

LISTING 13.7 An HTTP Request Containing a SOAP Method Call

```
1: POST /timeservice/timeweutils.asmx HTTP/1.1
2: Host: localhost
3: Content-Type: text/xml; charset=utf-8
4: Content-Length: 484
5: SOAPAction: "http://tempuri.org/webservices/GetTime"
6:
7: <?xml version="1.0" encoding="utf-8"?>
8: <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9:                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
10:                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
11:      <soap:Body>
12:        <GetTime xmlns="http://tempuri.org/webservices" />
13:      </soap:Body>
14:    </soap:Envelope>
```

Because SOAP uses HTTP, you can create a program that responds to SOAP requests using a Web server. This is why Web Services promise to become a popular way to create distributed programs. Currently, a huge number of Web servers are already in place, and given the right plumbing utilities, each one can be made to accept SOAP calls and return SOAP responses.

Because Web Services use SOAP and can run using Web servers, all .NET Web Services use ASP.NET as a foundation. ASP.NET is designed so that it can handle SOAP requests and dispatch them to your custom Web Services. If you create a Web Service using an ASP.NET page with an .asmx extension, ASP.NET will recognize it as a Web Service and dispatch SOAP requests to your Web server.



Note

As you might suspect, you can use SOAP without a Web server. By using .NET, you can also create server programs that use SOAP and communicate with client programs without the use of a Web server. This process, called *remoting* in .NET, is beyond the scope of this book. However, the documentation that comes with Visual Studio.NET contains detailed information and examples on how to set up servers using SOAP. Look under the Remoting keyword.

Finding out how to communicate with a remote Web Service is difficult unless there's a standard way to find out what methods the Web Service offers. As you saw earlier today, WSDL files supply this information. Any developer who needs to use a Web Service can look at the WSDL file and then build programs that call the remote Web Service. Of course, .NET automates this feature with the WSDL.exe tool and Visual Studio.NET.

The Web Services Vision

If the programming community adopts Web Services, expect to see many companies supplementing their Web sites with Web Services. They could include airline reservation services, music and book pricing and shipping services, and a host of other services that Web sites provide now.

You may be in a position to supplement your own company's Web or intranet site with a Web Service. For instance, if your company's intranet provides phone book and contact directories, you might implement a Web Service that provides the same information. This would allow other developers in your company to create desktop applications or other Web pages that access this contact information.

If you are a more ambitious developer, you may be contemplating a Web Service that provides a service to the general public, such as music CDs. Whatever your goal, the .NET framework provides all the tools you will need to create a Web Service of any size.

Summary

Today's lesson explained how to create and use Web Services. Now that you know the basics of Web Services and how they work, you are well equipped to learn more advanced techniques to use with Web Services, such as sending and returning custom data structures and ADO.NET datasets.

Today you learned how to create Web Service proxies with the .NET framework tools and automatically with Visual Studio.NET. You learned how a client program can use proxy class to call remote Web Services.

You also learned how Web Services communicate with client programs using the SOAP protocol. You saw examples of SOAP requests and learned why Web Services are used together with Web servers such as IIS.

Q&A

Q If I implement a Web Service using ASP.NET, will my service's clients be tied to the Microsoft platform?

A No. You can use any platform and language combination that supports TCP/IP. This makes your choice of client implementation almost unlimited. Client operating systems can include Windows 3.11 through Windows XP, all flavors of Unix and Linux, VAX, mobile device operating systems, and many others. Your choice of language to use for the client program could include Java, C, C++, Visual Basic, or any language that includes TCP/IP support.

Q How much about the SOAP protocol do I need to know to create Web Services?

A If you use .NET to create Web Services and their clients, you don't need to know much more than what you've seen in this chapter. ASP.NET and the .NET framework provide all the SOAP protocol plumbing for you. This situation is similar to the HTTP protocol and Web site programming, in that you can create most Web pages without detailed knowledge of HTTP requests and responses. To learn more about the SOAP protocol, visit [C:\mcp\BOOK0501\REVIEW\235813c2.dochttp://www.w3c.org](http://www.w3c.org).

Workshop

The workshop provides quiz questions to help you solidify your understanding of the material covered today as well as exercises to give you experience using what you have learned. Try to understand the quiz and exercise before continuing to tomorrow's lesson. Answers are provided in Appendix A, "Answers to Quizzes and Exercises."

Quiz

1. What file extension does ASP.NET use for Web Services?
2. Why do Web Services use WSDL files?
3. How do you retrieve the WSDL description for an ASP.NET Web Service?
4. Why should you use proxies for Web Service clients?
5. What files are created when you select Add Web Reference in Visual Studio?
6. Why do Web Services use ASP.NET?
7. Do all Web Services support HTTP Get and HTTP Post for calling functions?

8. List each standard in today's lesson that uses XML for its data format.
9. If a Web Service offers a method called `HelloWorld`, what two methods would a proxy program define for asynchronous calling?
10. What programming attribute do you use to make a method accessible to the public in a Web Service?

Exercises

1. Create a new method in the TimeUtils Web Service that returns the time of day in military and standard format. Have the method receive a `bool` parameter that specifies which format to use.
2. Create a Web Service using Visual Studio. What new files did Visual Studio create that we didn't in the TimeUtils example?
3. Create a Web Service that calls another Web Service. How could a Web Service like this be useful?

WEEK 2

DAY 14

Publishing Web Services

After you start to create your own Web Services, you will need a way to let the rest of the world know about them. Today's lesson will explain how clients discover Web Services and how to publish information about your Web Services.

The first section of today's lesson will explain exactly how the Add Web Reference dialog works in Visual Studio and will examine Web Service discovery files. These files can be used so that when Web Service consumers know the domain name of a Web server, they can easily find the complete specifications for all the services that the server provides.

Today's lesson will then introduce UDDI, an industry standard for listing Web Services with widespread support from many key leaders in the software industry. Even Microsoft and Sun, along with more than 80 major industry leaders, agree that UDDI should be the standard for Web Service discovery. You will learn about the structure and content of entries in the UDDI registry and see how to create entries for your own organization's Web Services.

Today's lesson covers the following topics:

- Dynamic and static Web Service discovery files and tips for how to choose between them
- UDDI, including an introduction and explanation of the UDDI registry

- Ways to add your own UDDI entries
- A sample UDDI entry

How Do Web References Work?

After yesterday's lesson, you may be wondering exactly how the Add Web Reference dialog works in Visual Studio. To refresh your memory, you display this dialog box by choosing Add Web Reference from the Project menu. You can use this dialog box to locate Web Services published by other companies; just click the [Microsoft UDDI Directory](#) link (you'll find more details on UDDI directories later today). Or, you can use it to list Web Services available on your local computer by clicking the [Web References on Local Web Server](#) link. How is this list of locally available Web Services obtained?

NEW TERM

The secret lies in what are called *discovery files*, or *disco files*. Visual Studio discovery files end in the extension .vsdisco. After .NET is installed on a computer, Internet Information Services (IIS) is configured so that it will hand off all .vsdisco files to ASP.NET whenever any application browses to one of these files.

So, what do discovery files do? They provide URL references to Web Services Description Language (WSDL) files, or to other discovery files. Recall from yesterday's lesson that WSDL files provide detailed information about a Web Service's public methods. By looking at a WSDL file for your Web Service, any program (or programmer) can create a Web Service proxy to use your Web Service. Discovery files provide, in XML format, the URL that any program (or programmer) needs to find a WSDL file for your Web Service. Discovery files can also provide the URL of another discovery file.

Discovery files, like almost everything else associated with Web Services, have an XML format. Each discovery file contains an XML document with elements that contain references to your Web Service's WSDL file. Listing 14.1 shows an example of a discovery file.

LISTING 14.1 Example1.vsdisco: A Sample Discovery File

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns="http://schemas.xmlsoap.org/disco/">
    <discoveryRef ref="http://localhost/WebBook/WebBook.vsdisco" />
    <discoveryRef ref="http://localhost/ComAspNet/ComAspNet.vsdisco" />
</discovery>
```

Three kinds of XML elements are used in discovery files: `discovery`, `discoveryRef`, and `contractRef`. Listing 14.1 shows the first two kinds. The `discovery` element is used as the root element in every static discovery file. The `discoveryRef` element gives a reference to another discovery file. Listing 14.1 contains references to two more discovery files, in the WebBook and ComAspNet virtual directories.

Listing 14.2 shows a discovery file that contains `contractRef` nodes.

LISTING 14.2 Example2.vsdico: Referencing WSDL Contracts

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns="http://schemas.xmlsoap.org/disco/">

    <contractRef ref="http://localhost/WebBook/AverageTiming.asmx?wsdl"
        docRef="http://localhost/WebBook/AverageTiming.asmx"
        xmlns="http://schemas.xmlsoap.org/disco/scl/" />

    <contractRef ref="http://localhost/WebBook/NorthwindService1.asmx?wsdl"
        docRef="http://localhost/WebBook/Timing.asmx"
        xmlns="http://schemas.xmlsoap.org/disco/scl/" />

</discovery>
```

The `contractRef` element references a WSDL file for a Web Service. It can also include a `docRef` attribute that points to an URL to find documentation about the Web Service. Because ASP.NET automatically generates documentation for a Web Service, the URLs for the Web Services themselves are given for the `docRef` attribute.

In general, each virtual directory that houses a Web Service also has a discovery file, which enables client programs to find the WSDL contract for the service. Also, the root virtual directory contains a discovery file with references to all the discovery files for each Web Service. Thus, by finding the discovery file in the root virtual directory, you can find all other disco files and the WSDL contracts for all available Web Services.

Web references in Visual Studio work by using discovery files. When you click the [Web References on Local Computer](#) link, Visual Studio opens the file default.vsdico in the root Web directory of your computer. At this point, ASP.NET takes over, processes the discovery file, and hands back an XML file containing references to all the Web Services on your computer. Visual Studio then uses its own version of the WSDL.exe tool to create a Web Service proxy, using the WSDL contract to which the discovery file points.

Dynamic Discovery Files

NEW TERM

The good news is that you don't have to create and maintain discovery files yourself. Instead, you can use a second flavor of discovery files, called *dynamic* discovery files.

The disco files covered in the preceding section are called *static* discovery files.

When a client program browses to a dynamic discovery file, the dynamic discovery file searches for WSDL contracts by itself and then returns an XML discovery document to the client. ASP.NET handles and creates the output for dynamic discovery files.

To make the discussion concrete, let's examine a dynamic discovery file. After a new .NET installation, a default.vsdisco file that uses dynamic discovery is put in the root of your Web server. Listing 14.3 shows what the default.vsdisco file looks like.

LISTING 14.3 Default.vsdisco: The Default Discovery Document in the Root of a Web Server

```
<?xml version="1.0" ?>
<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="_vti_cnf" />
<exclude path="_vti_pvt" />
<exclude path="_vti_log" />
<exclude path="_vti_script" />
<exclude path="_vti_txt" />
</dynamicDiscovery>
```

For dynamic discovery documents, ASP.NET searches through all the virtual directories on the Web server looking for more discovery (.vsdisco) files. It adds each discovery file that it finds to a list and outputs the results in a file that looks like a static discovery file. If you use your Web browser to look at the default.vsdisco file on your Web server root, you see something that looks like Listing 14.1.

Listing 14.3 contains XML elements labeled `<exclude>`. They instruct ASP.NET to avoid searching for more discovery files in the subdirectories named with the `path` attribute. By using the `<exclude>` element, you can hide directories from being discovered.

For dynamic discovery documents that reside in a virtual directory, ASP.NET searches for all Web Services in the directory (specifically for all files with an .asmx extension) and then outputs a static discovery file, such as Listing 14.1.

Preventing Dynamic Discovery

You may not want people browsing your Web server for all the services that it houses. You can easily disable dynamic discovery; to do so, just delete the .vsdisco file in the root virtual directory.

You can use these tips for the Web Service discovery issue:

- If you don't want the general public to find out what Web Services you have on your machine, remove the .vsdisco file from your Web server's root virtual directory.
- If you want to control which Web Services the general public can discover, replace the dynamic discovery files on your server with static discovery files, such as the code in Listing 14.1. Unfortunately, this option involves more work on your part because you have to maintain the discovery files yourself.
- To discover Web Services on a remote computer, browse for the discovery files or use the Add Web References dialog in Visual Studio.

What Is UDDI?

Today, finding a Web site that contains information that you need involves using a Web search engine. As you are no doubt well aware, finding Web sites this way isn't an easy task. Web search engines have become very sophisticated over the few short years that the Internet has been in popular use. Many Web search engines use a combination of sophisticated HTML parsing algorithms in combination with programs that automatically surf the Internet to gather the current information about Web sites. These search engines condense and index Web pages into large databases that you can browse using keywords or view by category. Unfortunately, Web search engines aren't perfect, and finding a Web site you need sometimes takes several search attempts. Even then, you still may never find the best Web site for the information you need.

Now that Web Services are appearing in greater numbers, the searching problem is even more difficult. Web Services rarely contain HTML pages that can be scanned for keywords. It doesn't take long to realize that Internet users need a new way to find information about organizations providing services on the Web.

UDDI (Universal Description, Discovery, and Integration) has been proposed as the solution to the problem of discovering Web Services. UDDI is a global registry that contains information about organizations and the public Web Services that they provide. UDDI is similar to an online yellow pages for Web Services.

As of this book's writing, three companies host the UDDI registry. Ariba, IBM, and Microsoft all share the hosting responsibilities. Each company periodically synchronizes the contents of the UDDI registry on its servers with the other two. This means that no single company controls the UDDI registry.

The UDDI project is still in its early stages. All companies contributing to the current UDDI project (currently more than 80) will hand off the UDDI concept to a standards body sometime in 2002. The home Web site for all information about UDDI is www.uddi.org.

Web Services and UDDI

NEW TERM UDDI uses the term *Web Service* in a broader way than today's lesson. A *UDDI Web Service* doesn't have to be a program that allows for remote method invocation using SOAP and XML. Instead, a UDDI Web Service can be as simple as a Web site that offers real estate listings as HTML pages. This sample real estate site is providing a service, after all.

The UDDI standard has three main goals:

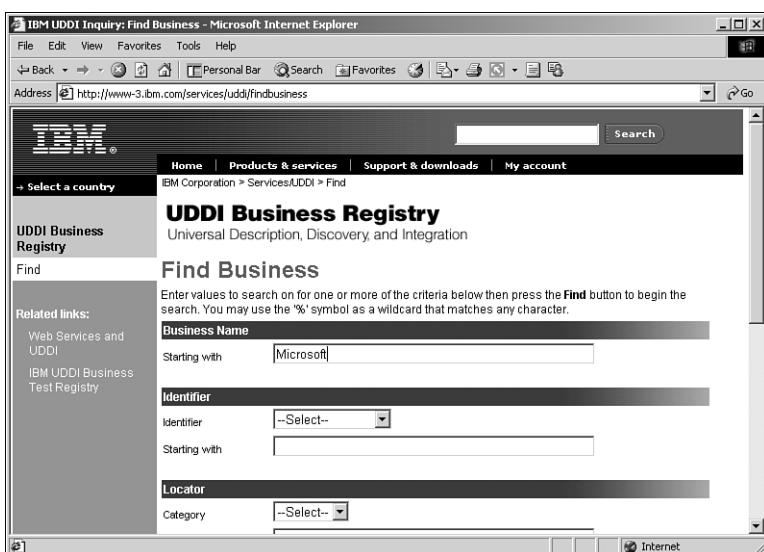
- Let consumers quickly find Web Services that serve a specific purpose.
- Provide a place that describes how Web Service transactions should take place.
- Create a standard way for businesses that offer Web Services to communicate with potential customers.

UDDI has been designed so that standard Web search engines can use it to supplement their own searching algorithms. Rather than find ad hoc methods to index Web Services, Web search engines can use a standard API to browse the UDDI registry.

However, you don't need to be a company such as Yahoo! to use UDDI. You can list all your company's Web Services in the UDDI registry by using a simple Web-based registration process, as you will see later today. You can search and browse the UDDI registry online. By using .NET, you can easily create programs that search the UDDI registry.

Let's look for some Web Services using IBM's Web site (www-3.ibm.com/services/uddi/findbusiness). You can also use the Microsoft UDDI directory, which is available directly from the Add Web Reference dialog. Both sites contain nearly identical features. Figure 14.1 shows the search page at the IBM site.

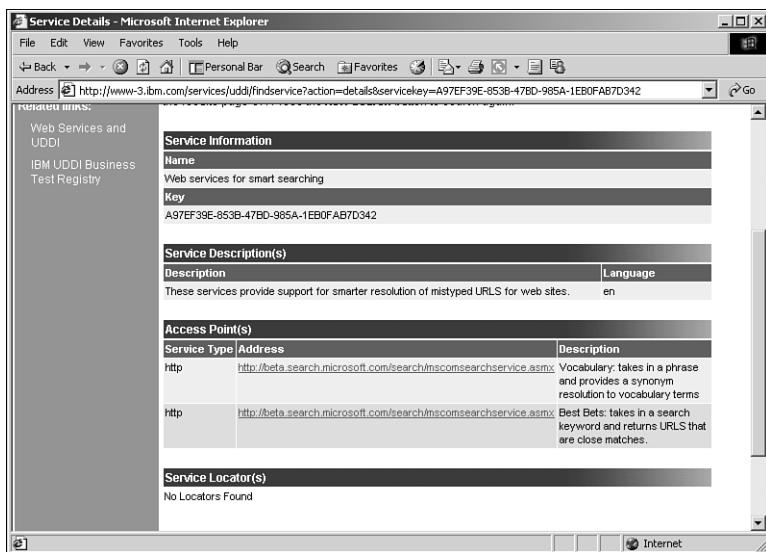
FIGURE 14.1
Searching for a business using a UDDI Web search site.



As an example, let's search for some of the Web Services that Microsoft has to offer. Enter **Microsoft** in the Business Name box. After you do so, the company listing appears, along with a link labeled services. Following this link displays a listing containing a number of Web Services (with *Web Service* defined in the most general sense), including links to the company's home page, online shopping, and many others. One link includes an entry called Web Services for smart searching. Clicking this link brings up a list of two Web Service pages, both of which link to the mscomsearchservice.asmx service. This list is shown in Figure 14.2.

FIGURE 14.2

A page containing links to public Web Services from a UDDI search site.



Creating Your Own UDDI Entry

If you create a public Web Service, listing it in the UDDI registry is one way to generate interest. Even if you don't have a Web Service ready yet, your company may provide general Web Services that you can list right now. You can easily list your own organization in the UDDI registry as follows:

1. Point your Web browser to <http://www.uddi.org/register.html>.
2. Choose from the list of UDDI registry provider companies. All companies share the same UDDI information among themselves, so your choice depends only on which Web site you prefer.
3. Create an account when the UDDI registry provider asks you to do so. Note that any information you give won't be available to the public.
4. Complete the Web site's easy-to-follow steps for adding a listing. This section won't give detailed instructions on this point. When you create your UDDI entry, you have the option of adding a technical contact. This contact person's information will be available to the public.

What's Inside a UDDI Entry?

UDDI entries can contain the following types of information:

- **A business name and description.** This information is required for every entry.
- **Technical contacts.** You can add any number of contacts for your organization.

- **Classifications for your organization's services.** UDDI uses a number of industry standard classification systems, such as the North American Industry Classification System and the Standard Industrial Classification. You should choose at least one classification for each service that your organization provides.
- **Classification bindings.** They are the actual references to your Web Services. They can include URLs to a Web site, URLs to a Web Service (in the specific sense), e-mail addresses, fax and phone numbers, or a reference of your own choosing.
- **A specification signature.** It describes the type of binding and can include entries such as `http://WSDL-interface` for a Web Service application. Other services, such as Web sites and e-mail addresses, don't have a specification signature.
- **Identifiers.** They identify your organization and can contain a D-U-N-S number.



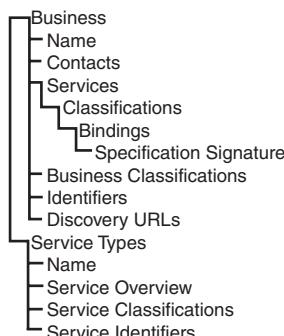
Note

Dun & Bradstreet identifies many organizations by using D-U-N-S® (Data Universal Numbering System) numbers. These numbers are used to verify the origin and status of a business. You can find out more about them at www.dnb.com.

- **Discovery URLs.** They contain URLs to Web pages giving more information about the services your organization provides.
- **Service types.** They contain descriptions of the services that your organization provides. Service types can contain classifications and identifiers (both discussed earlier in this list). Service types can also contain an overview document, which describes how the particular service type works, what kind of input it expects, and what kind of output it generates.

UDDI entries have a tree type format. To make the preceding list a little clearer, Figure 14.3 shows the structure of a UDDI entry.

FIGURE 14.3
The structure of a UDDI entry.



UDDI entries can contain quite a bit of information, and after a first glance at Figure 14.3, you might think it duplicates information. In particular, the Service Types node seems to duplicate the Services node in the Business node. The distinction is that the Service Types node contains specifications for Web Services. The Services node in the Business tree contains a reference to a specific Web Service. This specific Web Service is linked to a specification in the Service Types tree by using the Bindings node.

In particular, the Service Types entry contains a reference to the WSDL document for a Web Service that your organization provides. This WSDL reference is stored in the Service Overview node.

tModels and UDDI Entries

NEW TERM A *tModel* is a data structure (in XML, of course) that contains a reference to a WSDL file for a Web Service. When you create a Service Type (from Figure 14.3) using the UDDI provider's Web interface, you are creating a tModel for your Web Services. Other programs can find all your organization's tModels, use them to find the WSDL files for each of your Web Services, and then create proxy code to interact with your Web Service.

The idea of a tModel is more general than a specific WSDL file. One goal of the UDDI project is to build an industry standard set of tModels. This way, if your organization wants to provide a service to the public, you should implement a Web Service that corresponds to an industry standard tModel.

UDDI Entry Example

Let's examine what a specific UDDI entry would look like for a fictitious company called Bill's Discount Software. The lead software engineer at Bill's, named Sue, would like to create a UDDI entry for the company. She wants to create this entry because the Information Technology department has just created a new Web Service that allows client programs to receive and pay invoices. The Web Service, called SoftPay, is located at <https://www.billsdiscountsoftware.com/services/softpay.asmx>. The Web Service was designed to be used with a custom program called EZSoftwarePay. Bill's provides customers with the EZSoftwarePay client program for free, in hopes that the clients will use the program to pay their invoices quickly.

Sue first creates an account at one of the UDDI registry providers by following the links from www.uddi.org/register.html. This process takes a couple of days to complete because the UDDI provider needs to verify Sue's organization. After she receives an e-mail with a user ID and password, she logs into the provider site and begins to create an entry.

Sue has three main tasks:

- Create background information for her company in the UDDI entry.
- Optionally create a new Service Type if the Web Service her organization provides doesn't conform to an industry standard or if no industry standard service type is now defined for her organization's Web Service.
- Create a new Web Service definition for her organization and bind it to a Service Type.

Sue performs these steps to create background information for her company:

1. She adds the company name and description ("Bill's Discount Software" and "Provides software at a deep discount").
2. She adds two contacts for the company from the information technology department.
3. She adds the company's D-U-N-S number.
4. She adds a classification from the North American Industry Classification System (NAICS) for the business. She chooses Electronic Shopping and Mail-Order Houses from the Retail Trade group.



UDDI uses several standard classification systems for businesses, called *taxonomies*. They contain categories for most kinds of businesses. The NAICS taxonomy is one of several different classification systems available with UDDI.

Sue knows that her company is an industry leader in providing new Web Services, so for her second task, she wants to establish a new Service Type definition for the SoftPay service. She does so because she wants other companies to implement the SoftPay Web Service, in hopes that software invoicing can be automated through the industry. Sue follows these steps:

1. She adds a new Service Type called SoftPay with a description of Automatic Software Invoicing.
2. She adds an URL to the online technical documentation for developers who want to create their own Web Service that conforms to the SoftPay standard. This technical documentation contains links to WSDL contracts, in addition to detailed descriptions for how SoftPay services should work. This document URL is added to the Overview Document entry.
3. She adds a classification for the SoftPay service standard, using the NAICS classification system.

Last, Sue adds a new Web Service definition for her company. This Web Service definition will contain the URL for the company's implementation of the SoftPay service.

1. She adds a new service type, called Electronic Invoicing.
2. She adds a binding to the service to the URL
`http://www.billsdiscountsoftware.com/webservice/invoicing.asmx`.
3. She adds a specification type of SoftPay to the binding she just created.
4. She adds a classification for the service.

Figure 14.4 shows what the new Bill's Discount Software UDDI entry looks like.

FIGURE 14.4
Sample UDDI entry.



One benefit of the UDDI specification and infrastructure is that you can create your own programs to access a UDDI registry. You could create your own search engine to traverse the UDDI registry, find Web Services, and then connect to and use those services. Creating a program like this is an advanced topic beyond the scope of this book. However, you can use the Microsoft UDDI SDK to create .NET programs that access any publicly available UDDI registry. The SDK comes with a sample project that contains a program to browse UDDI registries and copy the contents of a registry onto your own machine. The SDK allows you to program against the UDDI API, which is defined at www.uddi.org. You can download the latest SDK from www.microsoft.com/downloads.

Summary

Today's lesson began by explaining discovery (disco) files and how you can use them to allow clients to browse all the Web Services available on a particular server. You saw two versions of discovery files: static and dynamic. Static discovery files contain XML documents that contain references to Web Services or other discovery files. Dynamic discovery files are handled by ASP.NET, which processes the file and returns a static discovery file as output.

The second half of today's lesson introduced the UDDI project and showed you how to use UDDI to publish Web Services in a global registry. You saw examples illustrating the structure of UDDI entries and a sample entry for a fictitious company. UDDI entries can define specific Web Services that an organization supports and can also be used to create new specifications for Web Services so that other organizations can create Web Services that implement this standard. You saw a sample Web Service standard called SoftPay, used for invoicing.

The next two lessons will dive into the nuts and bolts of creating real-world Web Services. You will learn how to pass complex data structures to Web Services in tomorrow's lesson, and Day 16, "Putting It All Together with Web Services," will give details on advanced Web Service programming techniques.

Q&A

Q I want to publish my Web Service. Should I use disco files or UDDI?

A Use both disco and UDDI. Disco files aren't listed to the Internet community at large but are useful for Web Service users when they know exactly what computer houses the Web Service. UDDI is a public record that anyone can use to find your Web Service. UDDI provides a great deal of supplementary information about a Web Service, including details about the organization providing the Web Service, contact information, and references to documentation.

Q Which UDDI provider should I use to publish my Web Service?

A Choosing a UDDI provider is only a matter of personal preference. All UDDI providers share the same database of information. After you make a UDDI entry in one provider, the entry propagates to all other UDDI providers within a short amount of time.

Q Is UDDI an industry standard now? If not, when will it become one?

A UDDI is presently an initiative that hasn't yet been ratified by a computer standards body. However, the UDDI initiative is scheduled to be handed over to a standards body in 2002.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered today as well as exercises to give you experience using what you've learned. Try to understand the quiz and exercises before continuing to tomorrow's lesson. Answers are provided in Appendix A, "Answers to Quizzes and Exercises."

Quiz

1. What's the difference between static and dynamic discovery files?
2. What is UDDI and what does it stand for?
3. How can you create a UDDI entry?
4. Name the two top-level nodes in a UDDI registry entry.
5. What's a tModel?

Exercises

1. Create a static discovery file for use with the TimeService example from yesterday's lesson.
2. Browse the www.uddi.org site for the latest developer documentation for the UDDI project. What's the complete definition of a tModel?
3. Download the Microsoft UDDI SDK. You will need to have SQL Server or MDAC installed on your computer first. Try using the sample .NET project to browse the Microsoft UDDI registry.

WEEK 3

DAY 15

Consuming a Web Service

Many Web Services that you write will contain methods that use custom data structures. For example, you may write a Web Service that stores an appointment schedule. Such a Web Service needs to have information about the name, start time, duration, and location of the appointment. A Web Service like this may need to receive complex data structures to be useful. Today's lesson will show how you can send and receive almost any kind of parameter to and from a Web Service.

After explaining how to define and use complex parameters, today's lesson will present a sample Web Service that uses the ADO.NET dataset object. This service will demonstrate some of the problems that can arise when more than one user tries to change the same data records at the same time. The example in today's lesson will offer one solution to the problem and give tips for solving the issue for your own projects.

Today you will learn about the following topics:

- How to define Web Service methods compatible with the HTTP protocol
- What data types can be used with the HTTP protocol

- What new types can be used with the SOAP protocol
- How to define custom data structures for use with a Web Service
- How to send and receive datasets to a Web Service
- How to deal with multiuser access to a Web Service
- How to use record-locking strategies in your own Web Services

Passing Parameters and Web Services

You may be wondering about what kinds of data structures you can send to a Web Service and the results that can be returned. This section will detail the different methods that you can use to pass parameters to Web Services and give several examples showing how to send complex parameters and receive complex parameters from Web methods.

Revisiting the Protocols Used for Web Services

Day 13, “Introducing Web Services,” gave only a small amount of detail concerning the Internet protocols that Web Services use to communicate. Understanding how to send and receive variables and objects to Web Services requires knowledge of a few key facts about Web Service protocols. This section will explain these protocols so that you are well equipped to learn about sending and receiving parameters in your own services.

.NET Web Services can use four different protocols without any extra work on your part: HTTP GET, HTTP POST, HTTPS, and SOAP. This section will explain each except for HTTPS, which will be explained on Day 19, “Securing Internet Applications.” To begin the discussion, let’s use the TimeUtil Web Service created on Day 13. It’s not repeated here, but you can refer to Listing 13.1 if you want to review the source code.

Let’s start by looking at a typical HTTP GET request and response, as shown in Listings 15.1 and 15.2.

LISTING 15.1 A Sample HTTP GET Request to a Web Service

```
GET /TimeService/TimeUtils.asmx/GetTime? HTTP/1.1
Host: localhost
```

So, what is an HTTP GET request? The request is just a stream of characters sent from a client program to a Web server, containing something like the text in Listing 15.1. Note that GET requests are limited to 1,024 characters.

Note

Try it yourself! Use Telnet to make an HTTP GET request by adding the following at the command-line prompt:

```
telnet localhost 80
```

Type the text in Listing 15.1 exactly and press Enter twice at the end. You can't see what you're typing, but you get the response shown in Listing 15.2.

15**LISTING 15.2** A Sample HTTP GET Response from a Web Service

```
1: HTTP/1.1 200 OK
2: Server: Microsoft-IIS/5.0
3: Date: Fri, 01 Aug 2001 21:48:21 GMT
4: Cache-Control: private, max-age=0
5: Content-Type: text/xml; charset=utf-8
6: Content-Length: 139
7:
8: <?xml version="1.0" encoding="utf-8"?>
9: <string xmlns="http://tempuri.org/">The current time on
10: the server is: 8/1/2001 5:48:21 PM</string>
```

ANALYSIS The GET response contains a standard header (Lines 1–6) and then the actual result from the Web Service call (Lines 8–10). HTTP POST requests are similar, with a slight change in the way parameters are passed in the request.

Now that you are well versed in the details of the GET request, you can understand one key limitation of using this protocol. By using HTTP, you can't send complicated parameters to the Web server. For a GET request, you can't do so because the request, including the parameter being passed in, has to fit on the first line of the request (first line of Listing 15.1). You might be able to convert a parameter into a string that would fit in the GET request. However, you could easily exceed the 1,024-character limit, and converting the parameter back and forth would be a messy operation.

In a nutshell, requests made to a Web server using the HTTP protocol are limited to the data types listed in Table 15.1.

TABLE 15.1 Parameters That Can Be Passed to a Web Service Using the HTTP Protocol

Data Type	Description
Primitive Types	char, int, long, float, double, Boolean, Int16, Int32, Int64, Single, Double, Currency, and unsigned versions of all numerical types
Strings	String
Enumerations	For example, enum myopinion {good, bad, indifferent}
Arrays	Arrays of any of the preceding types

As you saw in Listing 15.2, HTTP responses can contain any XML data, so there's no limitation on the parameters returned by using the HTTP protocol.

Sending and Receiving Parameters with SOAP

You can send *any* parameter you create in C# to and from a Web Service using SOAP—well, almost any. If the parameters that you create contain unmanaged code (code written using Visual C++, pre-.NET Visual Basic, assembly language, COM parameters, or parameters that you craft using the “hexadecimal” option in a text editor), they probably won't work. Parameters have to be transformed into XML format to work with SOAP. Luckily, every .NET class you create and all the built-in .NET framework classes can be transformed into XML. Furthermore, you don't have to transform the parameters yourself; the .NET classes in the `System.Web.WebServices.Protocols` namespace take care of this task automatically.

To see how a parameter is passed to a Web Service, look at the steps that happen when the client program invokes a Web Service method:

1. The client program calls a Web Service method using its Web Service proxy. Day 13 explained how to use Visual Studio .NET and the WSDL.exe tool to create Web Service proxies.
2. The proxy serializes each parameter into XML format.
3. The proxy creates a SOAP request using the XML version of the parameter.
4. The proxy opens a connection to the Web server hosting the Web Service.
5. The proxy transmits the SOAP request.

A reversed version of this process happens on the Web server so that the Web Service has a copy of the parameter that it can use.

Parameter Passing in Practice

You are now aware of the theory behind parameter transmission and Web Services. Let's put the theory into practice with some practical examples. To begin, let's create a Web Service that returns a complex parameter. The sample Web Service will extend the TimeUtils sample from Day 13 by adding a new method, `GetTimeEx`. This method returns a structure containing detailed information about the current time on the server.

First, create a definition for the parameter that the `GetTimeEx` method will return. Listing 15.3 shows the structure, `TimeStruct`.

LISTING 15.3 A Definition for the TimeStruct Structure, Which Will Be Returned by a Web Service

```
public struct TimeStruct
{
    public int Millisecond;
    public int Second;
    public int Minute;
    public int Hour;

    public String DayName;
    public String MonthName;
    public String YearName;

    public DateTime CurrentTime;
    public DateTime CurrentTimeUtc;
}
```

Listing 15.3 is a little contrived because the first seven member variables could easily be found using the `CurrentTime` variable. However, Listing 15.3 demonstrates one important point about passing parameters to and from Web Services: Any parameter that you want to pass to and from a Web Service must have a public class or struct definition. Also, any variable within the structure or class defining the parameter must be public.

Listing 15.4 shows the `TimeUtils` Web Service with the new `GetTimeEx` method.

LISTING 15.4 `TimeService.asmx`: Adding the `GetTimeEx` Method

```
1: using System;
2: using System.Web.Services;
3:
4: public struct TimeStruct
5: {
6:     public int Millisecond;
7:     public int Second;
8:     public int Minute;
9:     public int Hour;
10:
11:    public String DayName;
12:    public String MonthName;
13:    public String YearName;
14:
15:    public DateTime CurrentTime;
16:    public DateTime CurrentTimeUtc;
17: }
18:
```

LISTING 15.4 Continued

```
19: [WebService(Namespace="http://tempuri.org/webservices")]
20: public class TimeUtilities : WebService
21: {
22:     [WebMethod]
23:     public String GetTime()
24:     {
25:         String ret = "The current time on the server is: ";
26:         String curTime = DateTime.Now.ToString();
27:         return ret + curTime;
28:     }
29:
30:     [WebMethod]
31:     public TimeStruct GetTimeEx()
32:     {
33:         DateTime curTimeDT = DateTime.Now;
34:         TimeStruct curTime = new TimeStruct();
35:
36:         curTime.CurrentTime      = curTimeDT;
37:         curTime.CurrentTimeUtc  = curTimeDT.ToUniversalTime();
38:         curTime.Millisecond     = curTimeDT.Millisecond;
39:         curTime.Minute          = curTimeDT.Minute;
40:         curTime.Hour            = curTimeDT.Hour;
41:         curTime.DayName         = curTimeDT.DayOfWeek.ToString();
42:
43:         String monthAndDay = curTimeDT.ToString("m");
44:         String[] monthAndDaySplit = monthAndDay.Split(' ');
45:         curTime.MonthName = monthAndDaySplit[0];
46:
47:         curTime.YearName = curTimeDT.Month.ToString();
48:
49:         return curTime;
50:     }
51: }
```

ANALYSIS Lines 4–17 define the `TimeStruct` structure from Listing 15.3. Lines 19–28 are duplicated from Listing 13.1.

The `GetTimeEx` method (Lines 30–50) creates a new `TimeStruct` object (Line 34) and returns the populated object (Line 49). Lines 43–45 contain a small trick to find the name of the current month. First, the "`m`" format string returns the month and date (Line 43). A sample string returned from this format string might be "January 25". This string is split into two strings using the `Split` method (Line 44) so that the full month name can be found in the first element of the array that's returned (Line 45).

Note

Listing 15.4 doesn't take any special steps to return the structure; the code used here is similar to something you might write in any .NET application.

15

To complete the example, write a small client program to call the method. Listings 15.5 and 15.6 show a sample Web form and code behind files that call the Web Service. In a more realistic situation, the Web Service and Web page would be located on different server machines. If you have more than one computer at your disposal, try running the two listings separately.

Note

Listing 15.5 requires a Web Service proxy to work. This listing will work "as is" with a proxy generated by Visual Studio. Yesterday's lesson explained how to create Web proxies. As a quick reminder, you can use the following steps:

1. Make sure the source Web Service has been compiled without errors.
2. Choose Add Web Reference from the Project menu.
3. Enter the URL for the source service's WSDL file in the Address field.
An example might be `http://localhost/TimeService/TimeUtilities.asmx?WSDL`.
4. Click the Add Reference button to create the proxy.
5. Optionally, rename the Web reference, which is listed in the Solution Explorer under the Web References node.
6. In client code, make sure to include the Web proxy with the `using` keyword. If you named the proxy `TimeUtilitiesProxy`, add the line `using TimeUtilitiesProxy;` to the beginning of the client code file.

LISTING 15.5 DisplayTime.aspx: A Web Form That Uses Today's TimeUtils Web Service

```
<%@ Page language="c#" Codebehind="DisplayTime.aspx.cs"
Inherits="TimeService.DisplayTime" %>
<html>
<body>
    <h3>Current Time on the Server..</h3>
    <asp:Label id="Time" Runat="server" />
</body>
</html>
```

LISTING 15.6 DisplayTime.aspx.cs: The Code Behind File for Listing 15.5

```
1: using System.Web.UI;
2: using System.Web.UI.WebControls;
3: using TimeService.localhost;
4:
5: namespace TimeService
6: {
7:     public class DisplayTime : Page
8:     {
9:         protected Label Time;
10:        private void Page_Load(object sender, System.EventArgs e)
11:        {
12:            TimeStruct ts = new TimeStruct();
13:            TimeUtilities timeServer = new TimeUtilities();
14:            ts = timeServer.GetTimeEx();
15:            Time.Text = ts.DayName + " " + ts.MonthName + " " +
16:                        ts.CurrentTime.Day.ToString() + " , " +
17:                        ts.Hour.ToString() + ":" +
18:                        ts.Minute.ToString();
19:
20:        }
21:    }
22: }
```

ANALYSIS Listing 15.6 includes the namespace from the automatically generated proxy on Line 3. If you've used a different name to reference the Web server that contains the TimeUtils Web Service, you may have to change Line 3.

Lines 12–13 use the Web Service proxy to create a new `TimeStruct` and a new instance of the `TimeUtilities` proxy class. The code in Listing 15.5 connects to the Web Service when the `GetTimeEx` method is called (Line 14). After Line 14, the `TimeStruct` object will contain the values from the Web Service. Lines 15–18 set the `Label` control using the returned object.

As you can see from these listings, you don't have to write any special code to pass structures back from Web Services. The only tricky part of the process is generating the Web proxy class. You can rely on Visual Studio to create the proxy class simply by adding the Web reference.

Sending Parameters by Reference and by Value

NEW TERM Two other ways to pass parameters to a Web Service are known as *call by value* and *call by reference*. This section will explain what these terms mean and how they apply to Web Services.

Calling a method by value, or *byval* for short, means that when you pass a parameter into the method, the parameter will remain unchanged after the method is returned. Most programming languages make a copy of the parameter, pass this copied parameter into the method, and let the method change the parameter at will. When the method is done, the original parameter remains the same. The following code lines demonstrate the process:

```
void MyMethod(int intParam)
{
    intParam = intParam * 1000;
}
...
int i = 1;
MyMethod(i);
//i is still equal to 1 at this point
```

Calling a method by reference, or *byref* for short, means that the parameter passed into the method can be changed by the method, and the changes will “stick.” For instance, when the method changes all of a parameter’s variables, the parameter will keep the changes after the method is returned. Most programming languages implement this by passing in the original parameter to the method; no copies are made. Calling by reference is useful when you want a method to return multiple parameters. Simply pass in each parameter by reference, and use them as though they were return values from the method. This simple code shows the idea:

```
void MyMethod(ref int intParam)
{
    intParam = intParam * 1000;
}
...
int i = 1;
MyMethod(i);
//i is equal to 1000 at this point
```

Of course, the situation gets a little more complicated when Web Services are thrown into the mix. Passing a parameter to a Web Service by value involves making a copy of the parameter, persisting the parameter as XML, and passing the XML over the network to the Web Service.

Passing a parameter by reference to a Web Service uses the same method as the by value technique, with one extra step. After the method is finished, the same parameter is converted into XML by the Web Service and sent back to the client. The client program now has a modified parameter.



When you use the HTTP protocol (either GET or POST requests), all parameters must be passed by value, because HTTP doesn't support call by reference. In fact, the interface supplied by any ASP.NET Web service (the WSDL file) won't contain any by reference method definitions.

If you want to try passing a parameter by reference to a Web Service, add the following method to Listing 15.4:

```
[WebMethod]
public void GetTimeByRef(ref TimeStruct ts)
{
    ts = GetTimeEx();
}
```

Next, change Line 14 of Listing 15.6 to the following:

```
timeServer.GetTimeByRef(ts);
```

The modified listings should produce the same result as before. If you need to write a method that modifies more than one parameter, simply make each parameter byref by adding the `ref` keyword.

Accessing Data with Web Services

Because the SOAP protocol allows you to send any kind of parameter and receive any kind of return value, you can send and receive ADO.NET datasets. By using datasets in combination with Web Services, you can create Web Services that provide many kinds of data services. This section will describe a sample Web Service that manages a task list using dataset objects and XML files.

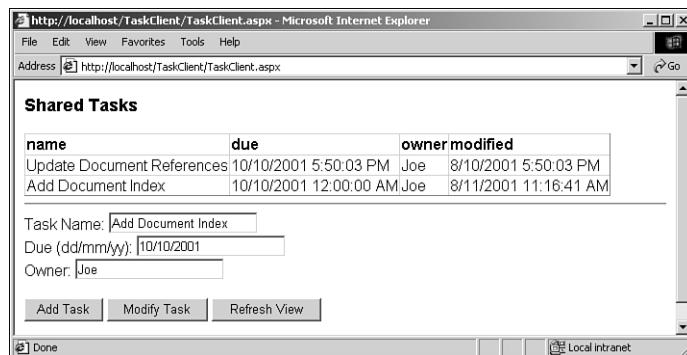
The Shared Task Web Service

The sample Shared Task Web Service allows multiple users to add and change tasks in a central location. The service stores the tasks in an XML file and allows any number of users to add and update tasks in the list. Figure 15.1 shows an example of a client that uses the Shared Task Web Service.

Any Internet application that allows more than one user to update data records has to deal with data concurrency issues. These issues consist of problems that can happen when two or more users update the same data record at the same time. You handle this problem by using a record-locking policy.

FIGURE 15.1

A Web form that uses the Shared Task Web Service.

**NEW TERM**

You can implement two kinds of record-locking policies to handle the simultaneous update problem: optimistic locking and pessimistic locking. *Optimistic locking* means that users can overwrite each other's changes. *Pessimistic locking* means that once a user obtains a lock on a data record, other users can't update that record until the first user releases the lock. Let's look at a scenario for each locking situation.

A typical pessimistic locking scenario might be the following:

1. User Joe reads a database record and sets a lock on the record.
2. User Sue reads the record but can't get a lock for it.
3. Sue tries to update the record but gets an error such as `The record is locked by Joe.`
4. Joe updates the record, saves changes, and releases the lock.
5. Sue can now update the record.

A typical optimistic locking scenario might be the following:

1. Joe reads a database record.
2. Sue reads the same record.
3. Sue updates the record.
4. Joe updates the record and overwrites Sue's changes.

From these scenarios, you might think that optimistic locking is a bad idea. Joe and Sue don't know that they overwrote each other's changes.

However, pessimistic locking can make a Web Service or Web application extremely slow because record locking takes up valuable database resources and can prevent other users from accessing other records. Most databases that use record locking must keep a

permanent database connection open to the client program to preserve the lock. Database connections are a scarce resource. If a server has too many open database connections, it will perform extremely slowly. To compound the problem, if your Internet application uses XML to store data, you have to implement your own record-locking logic, which can be quite complex.

One solution to the pessimistic locking problem involves using record versioning. One way of implementing record versioning is to attach a timestamp to each record. The scenario works like this:

1. Joe reads a database record. The record contains a field with a timestamp of the last time the record was modified.
2. Sue reads the same record, along with the same timestamp.
3. Sue updates the record. The timestamp is changed to the current time because the record has been modified.
4. Joe tries to update the record. His record's timestamp doesn't match the timestamp of the current record. The server detects this difference and gives Joe the option of refreshing his record to get the latest changes or simply overwriting the current record.

In this solution, Joe knows that he is about to overwrite a record whose value has changed.

Implementing Optimistic Locking

Let's implement the Shared Task Web Service so that it uses this record versioning solution to the optimistic locking program. Our first task is to design the look of the XML file that stores the shared tasks. Listing 15.7 shows an example.

LISTING 15.7 TaskStore.XML: A Sample XML File Containing Shared Tasks

```
<?xml version="1.0"?>
<sharedtasks xmlns="http://localhost/TaskServer/TaskStore.xsd">
  <task>
    <name>Cross Reference Documentation</name>
    <due>2001-10-05T12:00:00</due>
    <owner>Joe</owner>
    <modified>2001-08-10T17:50:03</modified>
  </task>
  <task>
    <name>Print Documentation</name>
    <due>2001-12-05T12:00:00</due>
    <owner>Sue</owner>
    <modified>2001-08-08T08:45:04</modified>
  </task>
</sharedtasks>
```

Each task contains a name, due date, owner name, and modified field.

Next, create a schema for the XML file. As a first step, use Visual Studio to automatically generate the schema. Then you can modify the schema to make the name field a primary key and to establish the appropriate data types for each field. The name and owner fields should be strings, and the due and modified fields should be of type `dateTime`. If you are following along with the lesson, you can make these changes in the Design view of the XML Schema editor in Visual Studio. The lesson for Day 11, “Using ADO.NET and XML Together,” gave details on how to use the editor. The final XML Schema should look like Listing 15.8.

LISTING 15.8 TaskStore.XSD: A Schema Definition for the Shared Tasks XML File

```
1: <xsd:schema id="sharedtasks_id"
2:                 targetNamespace="http://localhost/TaskServer/TaskStore.xsd"
3:                 xmlns="http://localhost/TaskServer/TaskStore.xsd"
4:                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5:                 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
6:                 attributeFormDefault="qualified"
7:                 elementFormDefault="qualified">
8:     <xsd:element name="sharedtasks"
9:                 msdata:IsDataSet="true"
10:                msdata:EnforceConstraints="true">
11:         <xsd:complexType>
12:             <xsd:choice maxOccurs="unbounded">
13:                 <xsd:element name="task">
14:                     <xsd:complexType>
15:                         <xsd:sequence>
16:                             <xsd:element name="name"
17:                                         type="xsd:string"
18:                                         minOccurs="0"
19:                                         msdata:Ordinal="0"
20:                                         msdata:AllowDBNull="false" />
21:                             <xsd:element name="due"
22:                                         type="xsd:dateTime"
23:                                         minOccurs="0"
24:                                         msdata:Ordinal="1" />
25:                             <xsd:element name="owner"
26:                                         type="xsd:string"
27:                                         minOccurs="0"
28:                                         msdata:Ordinal="2" />
29:                             <xsd:element name="modified"
30:                                         type="xsd:dateTime"
31:                                         minOccurs="0"
32:                                         msdata:Ordinal="3" />
33:                         </xsd:sequence>
34:                     </xsd:complexType>
35:                 </xsd:element>
```

LISTING 15.8 Continued

```
36:           </xsd:choice>
37:       </xsd:complexType>
38:       <xsd:unique name="task_con"
39:           msdata:PrimaryKey="true">
40:           <xsd:selector xpath=".//task" />
41:           <xsd:field xpath="name" />
42:       </xsd:unique>
43:   </xsd:element>
44: </xsd:schema>
```

ANALYSIS Lines 38–42 define a primary key for the XML file using the `unique` element. The name field is defined as the primary key field using the `selector` (Line 40) and `field` (Line 41) elements. Line 10 uses the `EnforceConstraints` attribute to make sure that ADO.NET uses the primary key when it reads the XML file. This way, ADO.NET will throw an exception if duplicated records with the same name field are added to the file.

Next, create a Web Service that allows users to add and modify tasks:

1. Create a new project using the Blank Web Project template. Name the project TaskServer.
2. Add a config.web file by choosing Add New Item from the Project menu.
3. Copy the TaskStore.xml file into the project directory. Add the file to the solution by choosing Add Existing Item from the Project menu.
4. Add the TaskStore.xsd file into the project.
5. Create a new typed dataset using the TaskStore.xsd file. To do so, select the TaskStore.xsd file from the Solution Explorer window. Then select the Generate DataSet option from the Schema menu.
6. Add a new Web Service file by using the Project menu's Add Web Service option. Call the Web Service TaskServer.asmx.

Create a method called `GetTasks` to return all tasks in the TaskStore.xml file to the user. Listing 15.9 shows what the method should look like.

LISTING 15.9 Returning a Dataset Containing All Shared Tasks

```
1: [WebMethod]
2: public sharedtasks GetTasks()
3: {
4:     sharedtasks ds = new sharedtasks();
```

LISTING 15.9 Continued

```
5:     ds.ReadXml(Server.MapPath("TaskStore.xml"));
6:     return ds;
7: }
```

ANALYSIS Line 2 declares that the `GetTasks` method should return an object of type `sharedtasks`. The `sharedtasks` class is defined in the typed dataset file that was just created in step 5. Lines 4–5 create a new typed dataset and read the content from the `TaskStore.xml` file. Line 6 returns the new typed dataset.

Now create a method that will add a new task. The method, called `AddTask`, should accept the `sharedtasks` typed dataset, which will contain one new task. The method should check to make sure that the new record doesn't have a duplicate name field.

Listing 15.10 shows an example of the `AddTask` method.

LISTING 15.10 Adding a New Shared Task

```
1: [WebMethod]
2: public bool AddTask(sharedtasks dsNewTask, ref String Error)
3: {
4:     bool bSuccess = true;
5:     sharedtasks.taskRow newRow = dsNewTask.task[0];
6:
7:     sharedtasks ds = GetTasks();
8:     try
9:     {
10:         ds.task.ImportRow(newRow);
11:         ds.WriteXml(Server.MapPath("TaskStore.xml"));
12:     }
13:     catch(ConstraintException ex)
14:     {
15:         bSuccess = false;
16:         Error = "A task with the name " + newRow.name + " already exists.";
17:     }
18:     catch(Exception ex)
19:     {
20:         bSuccess = false;
21:         Error = "Error: " + ex.Message;
22:     }
23:     return bRetVal;
24: }
```

ANALYSIS Line 5 creates a reference to the row containing the data for the new task called `newRow` by selecting the row at index 0 from the `task` table. Line 7 gets the current task list by calling the `GetTasks` method and assigns it the typed dataset called `ds`.

Lines 8–12 try to add the new row to the dataset by calling the `ImportRow` method (Line 10). `ImportRow` will throw a `ConstraintException` object if the new record contains a duplicate name field. Line 11 will write the new version of the shared task list back to disk if the `ImportRow` call was successful.

Lines 13–17 deal with a `ConstraintException`, if it occurs. Line 16 creates an error message that client programs can display to the user.

Next, add a method to the `TaskService` that will modify a row in the shared tasks file. This method needs to check the “modified” field of the record and match it against the “modified” field in the current record. If the two match, the client’s original data was valid. If the two modification times are different, the client’s data is out of sync with the data on the server. In this case, the method should return an error message. Listing 15.11 shows an example of the `ModifyTask` method.

LISTING 15.11 Modifying a Task, Checking to See Whether Another Client Has Modified the Record

```
1: [WebMethod]
2: public bool ModifyTask(ref sharedtasks dsTask, ref String Error)
3: {
4:     bool bSuccess = true;
5:
6:     sharedtasks.taskRow modifiedRow = dsTask.task[0];
7:
8:     sharedtasks ds = GetTasks();
9:     sharedtasks.taskDataTable taskTable = ds.task;
10:    DataRow[] matchingRows =
11:        taskTable.Select("name = '" + modifiedRow.name + "'");
12:
13:    if(matchingRows.Length == 1)
14:    {
15:        sharedtasks.taskRow rowToModify =
16:            (sharedtasks.taskRow) matchingRows[0];
17:
18:        if(rowToModify.modified == modifiedRow.modified)
19:        {
20:            modifiedRow.modified = DateTime.Now;
21:            ds.Merge(dsTask);
22:            ds.WriteXml(Server.MapPath("TaskStore.xml"));
23:        }
24:        else
25:        {
26:            bSuccess = false;
27:            Error = "This task has been changed by another user. " +
28:                   "Please refresh your view of the data.";
29:        }
}
```

LISTING 15.11 Continued

```
30:    }
31:    else
32:    {
33:        bSuccess = false;
34:        Error = "No task named " + dsTask.task[0].name + " found.";
35:    }
36:    return bSuccess;
37: }
```

ANALYSIS Line 6 creates a reference to the modified row that the client sent, called `modifiedRow`. Line 8 gets the current set of shared tasks by calling the `GetTasks` method.

Lines 10–11 return an array of `DataRow` objects in the shared task database that have the same name as the modified row that the client sent. Presumably, this array of matching rows contains only one value.

Lines 15–16 create a reference to the target row to be modified in the shared task database by retrieving the first object from the array of matching rows created previously.

Line 18 contains the key check to compare the modification timestamps between the client's record and the server's record. If they are identical, the client's data is up-to-date, and the changes should be accepted. Lines 20–22 update the server's version of the data and persist the data to disk.

Lines 24–30 create an error message if the client's data is out of sync with the server's version of the data.

As a final step, compile the new service.

Implementing a Client Web Form

This section will give an example of a Web form client that accesses the shared task service. You can use this Web form project to create multiple clients to the service and simulate the data concurrency problems that can occur. To create the client project, follow these steps:

1. Add a new project to the current solution using the Blank Web Project template (choose New Project from the File menu). Name the project `Task Client`.
2. Add a `config.web` file by using the Project menu's Add New Item option.
3. Add a new Web reference from the `TaskService` by using the Add Web Reference option on the Project menu.

4. Rename the proxy class created by Visual Studio to TaskServerProxy. To do so, right-click the localhost Web reference in the Solution Explorer window and select Rename.
5. Add a new Web form by using the Add Web Form option on the Project menu. Call the Web form TaskUI.aspx.

Next, create a user interface for the Web form. The form should contain a DataGrid to display the shared tasks and three buttons to add records, modify records, and refresh the grid. Also, add text boxes so that users can specify the name, due date, and owner of each new or modified record. Listing 15.12 shows an example of the Web form.

LISTING 15.12 TaskUI.aspx: A Web Form Client for the Task Service

```
1: <%@ Page language="c#" Codebehind="TaskUI.aspx.cs"
2:                         Inherits="TaskClient.TaskUI" %>
3: <html>
4:   <body>
5:     <font face="arial">
6:       <form runat="server" ID="Form1">
7:         <h3>Shared Tasks</h3>
8:         <asp:DataGrid ID="taskGrid" Runat="server"
9:                         HeaderStyle-Font-Bold="True"/>
10:        <hr>
11:        Task Name: <asp:TextBox Runat="server" ID="Name" /><br>
12:        Due (dd/mm/yy): <asp:TextBox Runat="server" ID="Due" /><br>
13:        Owner: <asp:TextBox Runat="server" ID="Owner" /><br>
14:        <br>
15:        <asp:Button Runat="server" OnClick="AddTask"
16:                      Text="Add Task" />
17:        <asp:Button Runat="server" OnClick="ModifyTask"
18:                      Text="Modify Task" />
19:        <asp:Button Runat="server" OnClick="OnRefresh"
20:                      Text="Refresh View" />
21:        <br>
22:        <asp:Label Runat="server" ID="ErrMsg" ForeColor="Red" />
23:      </form>
24:    </font>
25:  </body>
26: </html>
```

ANALYSIS Lines 8–9 define a data grid for the tasks, called taskGrid. Lines 11–13 define new text box controls that allow users to specify the task name, due date, and owner. Lines 15–20 define button controls to add tasks, modify tasks, and refresh data. Using the data grid for task editing could enrich this interface, but this example keeps things simple.

Next, set up the infrastructure for the client Web form. The client should keep its version of the task list in session state and contain methods to load the grid from the Web Service on the first page hit. Listing 15.13 shows the infrastructure for the Web form.

LISTING 15.13 TaskUI.aspx.cs: The Infrastructure Code for the Web Form Client

```
1: using System;
2: using System.Data;
3: using System.Web.UI;
4: using System.Web.UI.WebControls;
5: using TaskClient.SharedTaskProxy;
6:
7: namespace TaskClient
8: {
9:     public class TaskUI : System.Web.UI.Page
10:    {
11:        protected DataGrid taskGrid;
12:        protected Label ErrMessage;
13:        protected TextBox Name;
14:        protected TextBox Owner;
15:        protected TextBox Due;
16:
17:        private void Refresh()
18:        {
19:            TaskService ts = new TaskService();
20:            sharedtasks ds = ts.GetTasks();
21:            ds.EnforceConstraints = false;
22:            Bind(ds);
23:        }
24:        private void Bind(sharedtasks ds)
25:        {
26:            Session["Data"] = ds;
27:            taskGrid.DataSource = ds.task;
28:            taskGrid.DataBind();
29:        }
30:        protected void Page_Load(Object Sender, EventArgs e)
31:        {
32:            if(!IsPostBack)
33:            {
34:                Refresh();
35:            }
36:            else
37:            {
38:                sharedtasks ds = (sharedtasks) Session["Data"];
39:                Bind(ds);
40:            }
41:        }
42:        protected void OnRefresh(Object sender, EventArgs e)
43:        {
```

LISTING 15.13 Continued

```
44:     Refresh();
45: }
46: }
47: }
```

ANALYSIS The Refresh method (Lines 17–23) is responsible for fetching the latest version of the task list from the Web Service. The Bind method (Lines 24–29) binds the taskGrid Web control to a dataset and stashes the dataset in session state.

The Page_Load method (Lines 30–41) will either refresh or rebind the data grid, depending on whether this is the first time the page has been rendered or if this request is from a control postback.

The OnRefresh method (Lines 42–45) reloads the data grid by calling the Refresh method. The Refresh button, defined in the Web form user interface, calls this method.

The final two steps involve creating event handlers for the Add and Modify buttons. First, create the AddTask event handler, which should use the contents of the text boxes in the user interface to construct a new dataset. It should pass this dataset to the Web Service by calling the Web Service's AddTask method. Listing 15.14 shows an example of the client's AddTask method.

LISTING 15.14 Adding a New Task by Calling the Remote Web Service

```
1: protected void AddTask(Object Sender, EventArgs e)
2: {
3:     sharedtasks ds = (sharedtasks) Session["Data"];
4:     ds.AcceptChanges();
5:
6:     sharedtasks.taskDataTable dt = ds.task;
7:     dt.AddtaskRow(Name.Text, DateTime.Parse(Due.Text),
8:                   Owner.Text, DateTime.Now);
9:     sharedtasks delta =
10:        (sharedtasks) ds.GetChanges(DataRowState.Added);
11:
12:    TaskService ts = new TaskService();
13:    String Error="";
14:    bool bOk = ts.AddTask(delta, ref Error);
15:
16:    if(bOk)
17:    {
18:        ds.AcceptChanges();
19:        ErrMessage.Text = "";
20:    }
```

LISTING 15.14 Continued

```
21: else
22: {
23:     ds.RejectChanges();
24:     ErrMessage.Text = Error;
25: }
26: Bind(ds);
27: }
```

ANALYSIS Lines 3–4 get the contents of the client’s version of the task list from session state and call the `AcceptChanges` method to prepare the dataset for the new changes that are about to occur.

Lines 6–8 add a new row to the dataset, using the contents of the text boxes in the Web form. Lines 9–10 create a new dataset object, called `delta`, containing just this changed row by calling the `GetChanges` method.

Lines 12–14 invoke the `AddTask` method from the Shared Task Web Service, passing the `delta` dataset.

If the Web Service successfully adds the task, the `AcceptChanges` method is called (Line 18) so that the client’s version of the dataset reflects this new record. If the task can’t be added, Line 23 calls `RejectChanges`, which prevents the new record from being added to the dataset’s client copy.

The last method that needs to be added to the code behind file should handle record modifications. This event handler, `ModifyTask`, should modify the due date and owner fields only. The Web Service will take care of modifying the timestamp associated with the record. If the client’s original copy of the record is synchronized with the server’s record, the server will update the record’s timestamp to reflect the change. If another user has changed the record, the server won’t update the timestamp but will return an error message instead. Listing 15.15 shows an example of the `ModifyTask` event handler.

LISTING 15.15 Modifying a Task by Calling the Remote Web Service

```
1: protected void ModifyTask(Object Sender, EventArgs e)
2: {
3:     sharedtasks ds = (sharedtasks) Session["Data"];
4:     ds.AcceptChanges();
5:
6:     DataRow [] matchingRows = ds.task.Select("name = '" + Name.Text + "'");
7:     if(matchingRows.Length == 0)
8:     {
9:         ErrMessage.Text = "No task with this name exists";
```

LISTING 15.15 Continued

```
10:     return;
11: }
12: sharedtasks.taskRow modifiedRow =
13:     (sharedtasks.taskRow) matchingRows[0];
14:
15: //don't change modified time - server will do this on success;
16: //add stuff for get changes.
17: modifiedRow.due = DateTime.Parse(Due.Text);
18: modifiedRow.owner = Owner.Text;
19:
20: sharedtasks delta =
21:     (sharedtasks) ds.GetChanges(DataRowState.Modified);
22: String Error="";
23: TaskService ts = new TaskService();
24: bool bOk = ts.ModifyTask(ref delta, ref Error);
25:
26: if(bOk)
27: {
28:     modifiedRow.modified = delta.task[0].modified;
29:     ds.AcceptChanges();
30:     ErrMessage.Text = "";
31: }
32: else
33: {
34:     ds.RejectChanges();
35:     ErrMessage.Text = Error;
36: }
37: Bind(ds);
38: }
```

ANALYSIS Lines 6–13 contain code to find the row in the local dataset to modify by matching what the user entered into the name text box with records in the local dataset.

Lines 17–18 modify the fields in the row to match what the user entered in the Due and Owner text boxes.

Lines 20–21 create a new dataset, called `delta`, that contains the modified records. Lines 22–24 send the `delta` dataset to the Web Service by calling the service's `ModifyTask` method.

If the modification proceeds without any errors, the timestamp for the modified record is updated (Line 28), and Line 29 calls the `AcceptChanges` method. If the server returns an error, the `RejectChanges` method is called on Line 34.

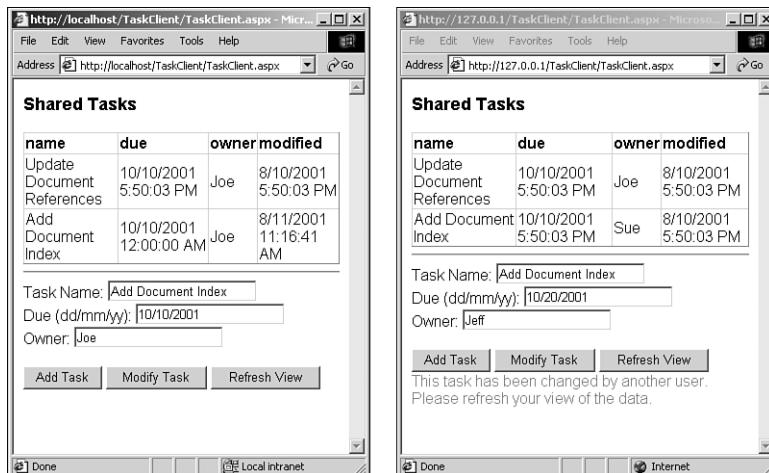
Trying Out the Client

To simulate two different users making changes to the same record, start two Web browsers. In the first Web browser, use the URL `http://localhost/TaskClient/TaskUI.aspx`. In the second browser, use the URL `http://127.0.0.1/TaskClient/TaskUI.aspx`. Using these different URLs has the effect of creating two different ASP.NET sessions, so you can simulate two different users.

Try adding and modifying records using the two different clients. Figure 15.2 shows an example in which one user tries to modify a record changed by another user.

FIGURE 15.2

Two shared task clients that modify the same record.



Summary

Today's lesson showed how to create Web Services that use and return custom data structures and complex parameters. The first section detailed each kind of data type that you can use with HTTP and SOAP. You saw how HTTP restricts many of your options for a Web Service and how you can use SOAP to transmit almost any kind of data.

The second half of today's lesson gave an advanced example of a Web Service that handles conflicting data modification requests from multiple clients. You learned the difference between optimistic and pessimistic locking and saw one strategy for overcoming some of the limitations of the optimistic locking scheme. The example used a record versioning strategy, which added a timestamp to each record. You can use the techniques presented in this sample in your own Web Service applications. These record-locking techniques aren't unique to Web Services and can be applied in many kinds of applications.

Q&A

Q Because the HTTP protocol is limited to call by reference, what practical effect does this have for my Web Service clients?

A Web Service clients that use HTTP instead of SOAP won't be able to use any Web Service methods that use call by reference parameters. ASP.NET generates the WSDL interface file automatically, and it won't generate method definitions for methods that use reference parameters for the HTTP protocol. All methods will contain definitions for the SOAP protocol.

Q I'm not sure if I should use an optimistic or pessimistic locking scheme for my Web Service. What are the critical issues?

A Optimistic locking works well when a large number of client programs don't modify the same records frequently. This situation can be described as the "low contention" scenario because client programs don't contend to modify the same records too often. However, if multiple client programs update the same records frequently, optimistic locking doesn't work very well. Use pessimistic locking for this "high contention" scenario, when many users want to modify the same data record.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered today as well as exercises to give you experience using what you've learned. Try to understand the quiz and exercises before continuing to tomorrow's lesson. Answers are provided in Appendix A, "Answers to Quizzes and Exercises."

Quiz

1. Can the HTTP protocol support methods that use the ADO.NET dataset as a parameter? Why or why not?
2. What is an "out" parameter?
3. What is a "ref" parameter?
4. What is optimistic locking?
5. Why is optimistic locking used in Web applications and services?
6. What method can you use to check the validity of a record modification request when using optimistic locking?

Exercises

1. The Task Server contains a major defect if used by many clients simultaneously. The problem is that the `Update` method might change the contents of the source XML file while the `AddTask` method is being called. Use the `.NET Monitor` class to synchronize access to the XML file in the Web Service.
2. Change the Task Server to use a SQL Server database or a database server of your choice. How much of the code needs to be changed?

WEEK 3

DAY 16

Putting It All Together with Web Services

By reading the last three days' lessons, you've gained a good knowledge base to use when creating your own Web Services. Today's lesson will build on the last three and will teach you some more advanced techniques that you can use in your own Web Services.

Today's lesson will begin by explaining how to *manage state* in your Web Services. This issue arises when you need to store information about particular Web Service clients between the calls that they make to your service. You will learn how to use the built-in Application and Session objects within your Web Services. As you will see, managing state can get complicated depending on the capabilities of the client accessing your service. This lesson will give solutions to many of these complex issues.

Today's lesson will continue by introducing a new technique called *asynchronous calling*. Web Service clients can use this method to call Web Service methods that may take a long time to complete. Clients can create a request to execute a Web method, perform other work, and then be signaled when the method is complete. You will see two examples that use this technique, including a Web form-based client.

After today's lesson, you should be well equipped to create your own Web Service applications. The remainder of this book will focus on deployment and configuration issues in Web applications and Web Services.

Today you will learn how to

- Use ASP.NET session state and application state in your Web Services
- Work around problems associated with ASP.NET session cookies
- Use asynchronous calling techniques for Web Service clients
- Create Web application client programs that use asynchronous calling

Managing State in Web Services

This section will explain how to use ASP.NET session and application state with Web Services. There's no quicker way to start a debate among .NET developers than to ask the question, "How do I manage per-user data in my Web Service using sessions?" Opinions vary on the use of session state in Web Services. Some regard the combination of session state and Web Services to be a terrible approach because some Web Services—maybe even your own—are designed so that thousands of users can call the service's methods at the same time. A Web Service that makes airline reservations might fit this description. If a Web Service like this uses session state, thousands of `Session` objects will be created at the same time. The creation of all these `Session` objects could quickly consume all of a Web server's resources. Web Services like these probably need to manage per-user data differently. Day 21, "Creating Your Application Architecture," will detail Web application and Web Service architectural issues. Until then, let's postpone the debate on ASP.NET sessions and Web Services. For now, we will show you how to use session state without making any pronouncements regarding whether you should use it.

Using Session State

In the lessons covering Web applications, you saw examples that use the `Session` object to store data for Web forms. For a quick review, remember that you can store data associated with each specific user of your Web site in the `Session` object, with statements such as the following:

```
Session["User Name"] = "Joe Smith";
```

and

```
String userName = (String) Session["User Name"];
```

The `Application` object can be used similarly. The lesson for Day 3, "Using Web Forms," contained an example that used the `Application` object to store objects for a Web application.

You can also use the `Session` and `Application` objects when developing ASP.NET Web Services. By using the `Session` object, you can keep track of information about clients that access your Web Service. Listing 16.1 shows a sample Web Service that keeps track of the amount of time spent on the server in a method called `DoWork`.

LISTING 16.1 Timing.asmx.cs: Keeping Track of the Time Spent in the `DoWork` Method

```
1: using System;
2: using System.Threading;
3: using System.Web;
4: using System.Web.Services;
5:
6: namespace WebBook
7: {
8:     public class Timing : System.Web.Services.WebService
9:     {
10:         private void AddTime(DateTime start, DateTime end)
11:         {
12:             Object oTime = Session["TotalTime"];
13:             TimeSpan ts;
14:             if(oTime == null)
15:             {
16:                 ts = new TimeSpan(0, 0, 0, 0, 0);
17:             }
18:             else
19:             {
20:                 ts = (TimeSpan) oTime;
21:             }
22:             ts += end - start;
23:             Session["TotalTime"] = ts;
24:         }
25:
26:         [WebMethod(EnableSession = true)]
27:         public void DoSomeWork()
28:         {
29:             DateTime startTime = System.DateTime.Now;
30:             // do some work
31:             Thread.Sleep(1000);
32:
33:             DateTime endTime = System.DateTime.Now;
34:             AddTime(startTime, endTime);
35:         }
36:
37:         [WebMethod(EnableSession = true)]
38:         public String GetTimeSpent()
39:         {
40:             if(Session["TotalTime"] != null)
41:             {
42:                 return ((System.TimeSpan) Session["TotalTime"]).ToString();
43:             }
44:         }
45:     }
46: }
```

LISTING 16.1 Continued

```
44:         return new System.TimeSpan().ToString();
45:     }
46: }
47: }
```

ANALYSIS The AddTime utility method (Lines 10–24) adds time to a total stored in the Session object. The method takes two DateTime objects representing the start and end times of a method call and calculates the difference between them by using the TimeSpan object (Line 22). It adds the difference in time to the total time and stores the updated value in Session (Line 23). Lines 14–21 contain logic to make sure that the total time count exists in Session.

The DoSomeWork method (Lines 27–35) calls the Thread.Sleep method (Line 31) rather than performs any real work. It takes a snapshot of the current time on the server at the beginning and end of the method (Lines 29 and 33) and uses the AddTime method to update the current user's time count.

The GetTimeSpent method (Lines 38–47) returns a string containing the latest total time figure for the DoSomeWork method. Of course, this Web Service doesn't accurately reflect the amount of CPU time that the server spent executing the code. It reports only the amount of elapsed time that the DoSomeWork method took to execute.

Using the Timing Web Service works normally if it's accessed from a Web form client. We won't show a Web form client here. You could use the following few lines of code to exercise the Web Service within your own Web form:

```
Timing timingServ = new Timing();
timingServ.DoSomeWork();
timingServ.DoSomeWork();
timingServ.DoSomeWork();
Response.Write(timingServ.GetTimeSpent());
```

The result from this code might look like the following line:

```
00:00:03.0043200
```

This output shows that the Web server spent just over 3 seconds to execute the three DoSomeWork calls, as expected.

Let's test the Timing service by using a console application for a client. Listing 16.2 shows just such an application.

LISTING 16.2 CallTiming.cs: Calling the Timing Web Service, with Unexpected Results

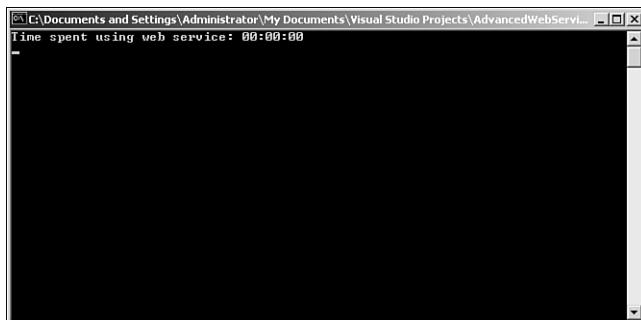
```
using System;
using System.Net;
using WebBook.localhost1;

namespace WebBook
{
    public class CallService
    {
        public static void Main()
        {
            Timing timingServ = new Timing();
            timingServ.DoSomeWork();
            timingServ.DoSomeWork();
            timingServ.DoSomeWork();
            Console.WriteLine("Time spent using web service: " +
                timingServ.GetTimeSpent());
        }
        Console.ReadLine();
    }
}
```

16

The output from Listing 16.2 looks like Figure 16.1.

FIGURE 16.1
*Calling a Web Service
using a console
application.*



Clearly, there is a problem here; the application should report a time of about 3 seconds.

To understand the problem, let's revisit how sessions work. ASP.NET normally uses browser cookies to keep track of sessions. When a new Web browser accesses a Web site or Web Service, ASP.NET generates a new empty `Session` object and new cookie for this browser. The cookie contains an ID number for the newly generated session. On each subsequent page hit from a browser, ASP.NET uses the ID number in the cookie to look up the user's session data, which is normally stored in memory in the Web server. ASP.NET takes the ID, looks up the session in memory, and then places that session into the `Session` object so that the Web Service (or Web form) can use it.

The problem with the Timing Web Service is that not all client programs support cookies. In fact, most clients, such as a Windows Forms application or console application, don't support cookies.

The solution to the problem involves a little trickery in two steps. First, stop ASP.NET from using cookies by changing one line of code in the web.config file.

**Note**

Day 3, "Using Web Forms," gave some introductory information on the web.config file, and Day 18, "Configuring Internet Applications," will explain all the options you can set in the file. As a quick review, you use the web.config file to change compilation settings, session state settings, debugging settings, and many more options. The web.config file should reside in the same directory as your Web Service.

You can use a web.config file such as the one in Listing 16.3 to stop ASP.NET from using cookies.

LISTING 16.3 A web.config File That Turns Off Cookies for Session Management

```
1: <configuration>
2:   <system.web>
3:     <compilation defaultLanguage="C#" debug="true" />
4:     <session cookieless="true" />
5:   </system.web>
6: </configuration>
```

ANALYSIS

Line 3 makes sure that debugging is enabled, and Line 4 turns off cookies.

NEW TERM

Now that cookies are disabled, ASP.NET will use a technique called *URL munging* to keep track of sessions. URL munging works by taking the ID for a session that's normally stored in a cookie and appending it to the URL for the Web Service or Web form. This means that on the first method call to the Web Service, the client will be redirected to a new "munged" URL, such as the following:

`http://localhost/WebBook/(ylotjcyo4hhe00nkqwmr1r55)/Timing.asmx`

Unfortunately, this new URL will cause an exception in the console application client when it invokes the Web proxy. The standard Web proxy doesn't work when the Web Service redirects it to a new URL.

At this point, the second part of the solution is necessary. You have to make a small modification to the client program so that it will go to the new URL after the first call to the Web Service. Listing 16.4 shows one way to implement the solution.

LISTING 16.4 CallTiming2.cs: Handling Redirection to a Munged URL

```
1: using System;
2: using System.Net;
3: using WebBook.localhost1;
4:
5: namespace WebBook
6: {
7:     public class CallService
8:     {
9:         public static void Main()
10:        {
11:            Timing timingServ = new Timing();
12:            //Make an attempt to call a method. This will result in
13:            //an exception, because the web server redirects us to a
14:            //"munged" url
15:            try
16:            {
17:                timingServ.DoSomeWork();
18:            }
19:            catch (WebException e)
20:            {
21:                //Handle the exception. The message of the exception will
22:                //contain a reference to the new URL
23:                String err = e.Message;
24:
25:                //Create a search string to find the new URL
26:                String search = "Object moved to <a href'";
27:                int newUrlPos = err.IndexOf(search);
28:
29:                if(newUrlPos > 0)
30:                {
31:                    //We found the magic string, so redirect ourselves
32:                    String rightString = err.Substring(newUrlPos + search.Length);
33:
34:                    //Split the string using the ' character. The second element
35:                    //will contain the new URL - the first and third elements
36:                    //contain the rest of the error message
37:                    String[] segmentedString = x.Split('\\');
38:
39:                    String newUrl = segmentedString[1];
40:
41:                    //Redirect ourselves by changing the URL in the
42:                    //web server proxy
43:                    timingServ.Url = "http://localhost" + newUrl;
44:                }
45:                else
46:                {
47:                    //this must be a different error
48:                    throw e;
49:                }
50:            }
51:        }
52:    }
53: }
```

LISTING 16.4 Continued

```
52:     timingServ.DoSomeWork();
53:     timingServ.DoSomeWork();
54:     timingServ.DoSomeWork();
55:     String timeSpent = timingServ.GetTimeSpent();
56:     Console.WriteLine("Time spent using web service: " + timeSpent);
57: }
58: }
59: }
```

ANALYSIS The new code to redirect the Web proxy to the munged URL is contained in the try...catch block in Lines 15–50. The catch block in Lines 19–50 searches for a “magic” string in the exception’s error message (Line 26–27). If the search string is found, the code parses the new URL (Lines 32–39) and then replaces the URL for the Web Service proxy (Line 43).

Using Application State

ASP.NET application state doesn’t have the complications that surround session state for Web Services because application state is independent of specific users. Application state doesn’t need to be tracked using cookies. Not needing to use cookies simplifies your Web Service and Web Service client code significantly.

For instance, you could modify Listing 16.4 slightly to record statistics about a Web Service. Rather than record information user by user about the call time for the DoSomeWork method, the Web Service could record times for every user. You then could modify the GetTimeSpent method to return the average time spent in the DoSomeWork method for all the service’s clients.

Listing 16.5 shows a sample Web Service that records average call times for a method.

LISTING 16.5 AverageTiming.asmx.cs: The Code Behind for a Web Service That Records Average Call Time for a Method

```
1: using System;
2: using System.Threading;
3: using System.Web;
4: using System.Web.Services;
5:
6: namespace WebBook
7: {
8:     public class AverageTiming : WebService
9:     {
10:         private void AddTime(DateTime start, DateTime end)
11:         {
```

LISTING 16.5 Continued

```
12:     int calls = 0;
13:     Object oTime = Application["TotalTime"];
14:     TimeSpan ts;
15:     if(oTime == null)
16:     {
17:         ts = new TimeSpan(0, 0, 0, 0, 0);
18:     }
19:     else
20:     {
21:         ts = (TimeSpan) oTime;
22:         calls = (int) Application["Calls"];
23:     }
24:     ts += end - start;
25:     Application["TotalTime"] = ts;
26:     Application["Calls"] = calls + 1;
27: }
28:
29: [WebMethod]
30: public void DoSomeWork()
31: {
32:     DateTime startTime = System.DateTime.Now;
33:     // do some work
34:     Thread.Sleep(1000);
35:
36:     DateTime endTime = System.DateTime.Now;
37:     AddTime(startTime, endTime);
38: }
39:
40: [WebMethod]
41: public String GetAverageTimeSpent()
42: {
43:     TimeSpan ts = (TimeSpan) Application["TotalTime"];
44:     int numCalls = (int) Application["Calls"];
45:     double avgTime = (ts.Seconds * 1000 + ts.Milliseconds)/numCalls;
46:     return avgTime.ToString();
47: }
48: }
49: }
```

16

ANALYSIS The AddTime method (Lines 10–27) has been changed to use the Application object. Also, the AddTime method records the total number of calls and stores this figure in the Application object (Lines 22 and 26). The AddTime method also records the total amount of time spent inside the calling method (Lines 24–25).

The DoSomeWork method (Lines 30–38) is unchanged from Listing 16.4. The GetAverageTimeSpent method (Lines 41–47) finds the average call time so far in milliseconds and returns the value.

Dealing with Slow Services

Web Service methods may take a long time to complete, especially if the method performs a complicated operation or must access other slow components. For instance, an airline reservation Web Service may have to wait on a legacy reservation system that takes a few seconds to complete. In this case, the client program may be blocked, waiting for the Web Service method to complete. This blocking behavior could make the client program appear to be frozen to users, and users might get impatient and stop the client program. The solution to this problem is to call Web Service methods asynchronously.

NEW TERM Calling a Web Service method *asynchronously* means that each method call is broken into two steps:

1. The client program generates a request to the Web Service. This request contains all the necessary parameters for the Web Service method. The client sends the request to the Web Service and, rather than wait for a response, performs other tasks, such as displaying a progress bar. When the client makes the request, it also gives the Web Service a reference to a callback method.
2. When the Web Service is finished, it returns the data, and the Web Service proxy invokes the callback method. The callback method can examine the results from the Web Service and process the results. For the airline reservation example, the client program can inform users about the status of their reservation requests.

If the Web Service doesn't return a result in time, the client can abort the connection and take steps to tell users that the method failed.

.NET has built-in support for asynchronous calling, and Web Services and Web Service proxies use the .NET classes. If you are experienced with asynchronous calling techniques in .NET, the following explanation for Web Services won't be any different from what you are used to already. If you've never implemented a program that uses asynchronous calling, don't worry; the rest of this section will give you step-by-step instructions.

The following steps outline how to create a program to call Web Service methods asynchronously. The following sections will explain each step in detail.

1. Create a callback method.
2. In the callback, make a call to the `EndXXX` proxy method, where `XXX` is the name of the Web method. This method will return the results from the Web Service method.
3. In the method that calls the Web Service, create an `AsyncCallback` object that wraps the actual callback method.

4. Invoke the Web Service method by using the `BeginXXX` proxy method, where `XXX` is the name of the Web method.
5. Write code to do other work or wait for the method to complete. You can abort the call at any time.

Let's see how to carry out each one of these steps. To simplify the task of learning about asynchronous calling, we'll use a console application as a client in this first example.

However, you can apply the same techniques to any kind of client application. At the end of this section, you'll create a Web form that uses asynchronous calling.

Of course, we need a Web Service that contains a slow method to see how this process works. Listing 16.6 shows just such an example.

16

LISTING 16.6 SlowService.asmx.cs: The Code Behind for a Slow Web Service

```
1: using System;
2: using System.Threading;
3: using System.Web;
4: using System.Web.Services;
5:
6: namespace WebBook
7: {
8:     [WebService]
9:     public class SlowService : WebService
10:    {
11:        [WebMethod]
12:        public String SlowMethod(int waitSec)
13:        {
14:            Thread.Sleep(waitSec * 1000);
15:            return "The answer is...43.";
16:        }
17:    }
18: }
```

ANALYSIS The `SlowMethod` (Lines 11–16) uses one parameter, `waitSec`, to determine how many seconds it should wait. This parameter allows you to set exactly how slow the method will be in a client program, which is useful for experimentation. The `SlowMethod` returns a string.

Creating a Callback Method

Creating a callback method is the first step to take for the “asynchronous” client. The code in Listing 16.7 shows a sample callback method.

LISTING 16.7 A Sample Callback Method

```
1: public void MyCallBack(IAsyncResult ar)
2: {
3:     try
4:     {
5:         String result = slowServ.EndSlowMethod(ar);
6:         Console.WriteLine("Callback:Web Service returned: " + result);
7:     }
8:     catch(Exception)
9:     {
10:        Console.WriteLine("Callback:The call was aborted!");
11:    }
12: }
```

ANALYSIS All callback methods must follow the pattern on Line 1. That is, the callback method can't return a value, and the method must take one parameter of type `IAsyncResult`. The `IAsyncResult` result object contains the result of the call to the Web method.

Line 5 contains the code to process the result object and extract the return value from the Web method. Notice that Line 5 uses the `EndSlowMethod` call from the Web proxy. Every Web proxy generated by Visual Studio (or WSDL.exe) contains a method like this. For each Web Service method, such as `ExcellentMethod`, there will be an `EndExcellentMethod` and `BeginExcellentMethod` in the Web proxy by default.

The callback method uses a `try...catch` block (Lines 3–11) in case the asynchronous call is aborted. The next section will explain how to abort asynchronous calls. When a method is aborted, the callback is invoked immediately, and the `EndSlowMethod` call throws an exception.

Calling the Web Method Asynchronously

Now that the callback is written, you need to create code to call the Web method asynchronously. As you might have guessed, you must use the `BeginSlowMethod` (or `BeginAppropriateMethod`) call to do so. Before doing so, you must take some preparatory steps:

1. Create a new `AsyncCallback` method that contains a reference to the callback you created in the preceding section. For this example, you use the following code line:
 `AsyncCallback callbackRef = new AsyncCallback(MyCallBack);`
2. Call the Web method using a line like the following:

```
IAsyncResult ar = slowServ.BeginSlowMethod(5, callbackRef, null);
```

ANALYSIS To understand the preceding code line, let's examine it from right to left. The `BeginSlowMethod` call takes three parameters:

- The value 5 corresponds to the first parameter that the original `SlowMethod` receives, so this Web method will take at least 5 seconds to complete.
- `callbackRef` is the `AsyncCallback` object created in the preceding section.
- The `null` parameter is used for extra information. This parameter should always be `null` for the current version of .NET.

The asynchronous call returns an `IAsyncResult` object, which you can use to get information about the asynchronous call's status. Listing 16.8 uses the asynchronous result object to wait on the method.

LISTING 16.8 Partial Code Sample to Wait for an Asynchronous Call by Polling the Result Object

```
1: Console.WriteLine("Waiting on the web service...");  
2: int totalWaitTime = 6000; //wait 6 seconds  
3: //int totalWaitTime = 1000; //Uncomment this line to be impatient  
4:  
5: //replace this with code to do some real work  
6: ar.AsyncWaitHandle.WaitOne(totalWaitTime, false);  
7:  
8: if(ar.IsCompleted)  
9: {  
10:   Console.WriteLine("The call completed!");  
11: }  
12: else  
13: {  
14:   Console.WriteLine("The call did not complete in time. Aborting...");  
15:   slowServ.Abort();  
16: }
```

ANALYSIS Line 2 sets the `totalWaitTime` variable to the total amount of time to wait in milliseconds. Line 6 calls the `WaitOne` method, which waits for the `IAsyncResult` object (the object named `ar`) to become set to a completed state. This is known as waiting for the object to be *signaled*. The `WaitOne` method will return as soon as the asynchronous call is complete or when the timeout period expires, whichever comes first.

Lines 8–16 test whether the method is completed in time by using the `IsCompleted` property. If the method isn't completed in time, the code calls the `Abort` method (Line 15) to stop the asynchronous call immediately.

Completing the Asynchronous Client

Listing 16.9 contains the complete code for the console application client.

LISTING 16.9 A Client That Calls a Web Service Asynchronously

```
1: using System;
2: using System.Threading;
3:
4: using Proxy; //The namespace of the automatically generated proxy
5:           //change this to the namespace for the web proxy you
6:           //have generated
7: namespace WebBook
8: {
9:     public class CallAsynch
10:    {
11:        public static void Main()
12:        {
13:            WebServiceCaller obj = new WebServiceCaller();
14:            obj.Call();
15:            Thread.Sleep(50);
16:        }
17:    }
18:    public class WebServiceCaller
19:    {
20:        SlowService slowServ;
21:        public void MyCallBack(IAsyncResult ar)
22:        {
23:            try
24:            {
25:                String result = slowServ.EndSlowMethod(ar);
26:                Console.WriteLine("Callback:Web Service returned: " + result);
27:            }
28:            catch(Exception)
29:            {
30:                Console.WriteLine("Callback:The call was aborted!");
31:            }
32:        }
33:        public void Call()
34:        {
35:            slowServ = new SlowService();
36:            AsyncCallback cb = new AsyncCallback(MyCallBack);
37:
38:            IAsyncResult ar = slowServ.BeginSlowMethod(5, cb, null);
39:
40:            Console.WriteLine("Waiting on the web service...");
41:            int totalWaitTime = 6000; //wait 6 seconds
42:            //totalWaitTime = 1000; //Uncomment this line to be impatient
43:
44:            ar.AsyncWaitHandle.WaitOne(totalWaitTime, false);
```

LISTING 16.9 Continued

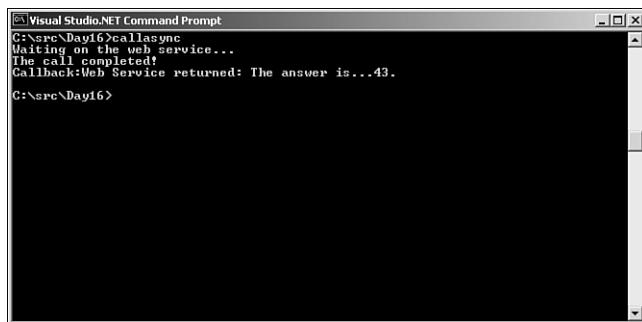
```
45:  
46:     if(ar.IsCompleted)  
47:     {  
48:         Console.WriteLine("The call completed!");  
49:     }  
50:     else  
51:     {  
52:         Console.WriteLine("The call did not complete in time." +  
53:             " Aborting...");  
54:         slowServ.Abort();  
55:     }  
56: }  
57:  
58: }
```

16

After running this sample, you should see output similar to Figure 16.2.

FIGURE 16.2

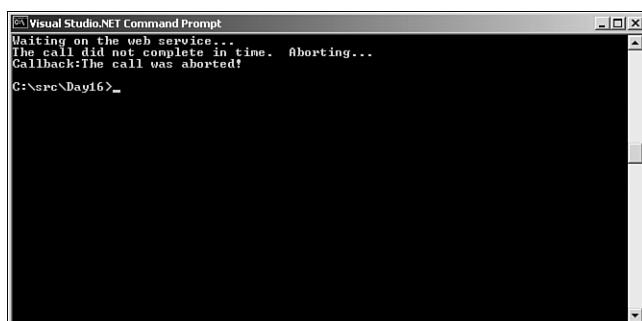
*Calling a Web Service
using an asynchronous
client console
application.*



If you uncomment Line 42, recompile, and then rerun the client, you should see output similar to Figure 16.3.

FIGURE 16.3

*An aborted
asynchronous call
to a Web Service.*



ANALYSIS

The preceding discussion already detailed most of the code in Listing 16.9. However, one line is worth noting. Line 15 contains a `Sleep` call after the `Call` method. This line is optional and was inserted into this sample so that you can see the result from the callback when the asynchronous call is aborted. Without the delay, the program would exit before the callback had a chance to complete. Removing the delay won't break the example; the program will still complete gracefully.

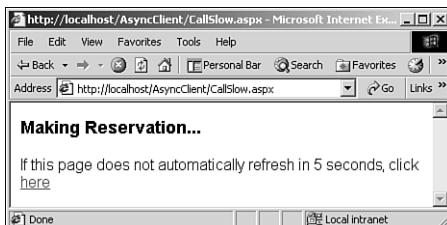
Listing 16.9 is fine for demonstration purposes, but a console application probably isn't your first choice for a Web Service client. Let's use the concepts in Listing 16.9 to create a Web form client for the `SlowService` Web Service in Listing 16.6. This client will model a request for an airline reservation, using `SlowService` as the back-end "reservation system."

The client will work using the following steps:

1. The Web form will display the message `Making Reservation...`, similar to Figure 16.4.

FIGURE 16.4

A Web form that calls a Web Service asynchronously.



2. In the `Page_Load` event, the Web form will generate an asynchronous call to the `SlowService` Web Service.
3. The Web form will contain HTML code so that the browser will automatically refresh after 5 seconds.
4. The `Page_Load` event will contain code to detect whether it has been called a twice. If so, it will check for a result. If the asynchronous call is completed, the results will be shown. If not, the Web form will display a `Still waiting...` message.

Listing 16.10 shows a Web form to call the `SlowService` Web Service, and Listing 16.11 shows the code behind file that calls a Web Service asynchronously using the four preceding steps.

LISTING 16.10 CallSlow.aspx: A Prototype Reservation Page that Calls a Web Service Asynchronously

```
1: <%@ Page language="c#" Codebehind="CallSlow.aspx.cs"
2:     Inherits="WebBook.CallSlow" %>
3: <html>
4: <head>
5:     <meta http-equiv='refresh' content='5'
6:         id="metaTag" runat="server"></meta>
7: </head>
8: <body>
9:     <font face="arial">
10:        <h3><asp:Label ID="Message" runat="server"/></h3>
11:        <div id="refreshMessage" runat="server">
12:            If this page does not automatically refresh in 5 seconds,
13:            click <a href="CallSlow.aspx">here</a>
14:        </div>
15:    </font>
16: </body>
17: </html>
```

ANALYSIS Lines 5–6 contain a `<meta>` tag that tells the browser to refresh the current page after 5 seconds. This `<meta>` tag is defined as a server control so that it can be disabled by the code behind file after the reservation is completed successfully. Lines 11–14 contain a `<div>` tag that's used exactly the same way as the `<meta>` tag. The `<div>` tag will be hidden by the code behind file after the reservation is completed.

LISTING 16.11 CallSlow.aspx.cs: The Code Behind File for Listing 16.10

```
1: using System;
2: using System.Web.UI;
3: using System.Web.UI.WebControls;
4: using System.Web.UI.HtmlControls;
5: using System.Web.Services.Protocols;
6:
7: using SlowProxy; //namespace for the proxy class. This may be
8:                     //different in your own project
9: namespace WebBook
10: {
11:     public class CallSlow : System.Web.UI.Page
12:     {
13:         protected Label Message;
14:         protected HtmlGenericControl refreshMessage;
15:         protected HtmlGenericControl metaTag;
16:
17:         protected void MyCallBack(IAsyncResult ar)
18:         {
19:             Session["Result"] = ar;
20:         }
```

LISTING 16.11 Continued

```
21:      protected void Call()
22:      {
23:          Message.Text = "Making Reservation...";
24:
25:          SlowService slowServ= new SlowService();
26:          AsyncCallback cb = new AsyncCallback(MyCallBack);
27:          slowServ.BeginSlowMethod(4, cb, null);
28:
29:
30:          Session["CalledMethod"] = true;
31:      }
32:
33:  private void Page_Load(object sender, System.EventArgs e)
34:  {
35:      if(Session["CalledMethod"] == null)
36:      {
37:          Call();
38:      }
39:      else if (Session["Result"] == null)
40:      {
41:          Message.Text = "Still waiting....";
42:      }
43:      else
44:      {
45:          metaTag.Visible = false;
46:          refreshMessage.Visible = false;
47:
48:          if(Session["ReservationMade"] == null)
49:          {
50:              try
51:              {
52:                  SlowService slowServ = new SlowService();
53:                  IAsyncResult ar = (IAsyncResult) Session["Result"];
54:                  String result = slowServ.EndSlowMethod(ar);
55:
56:                  Message.Text = "Successfully Made Reservation!";
57:                  Session["ReservationMade"] = true;
58:              }
59:              catch(Exception ex)
60:              {
61:                  Message.Text = ex.Message;
62:              }
63:          }
64:          else
65:          {
66:              Message.Text = "Your reservation has been made.  " +
67:                            "Please return to the XYZ page if you would " +
68:                            "like to make another.";
69:          }
}
```

LISTING 16.11 Continued

```
70:      }
71:    }
72:  }
73: }
```

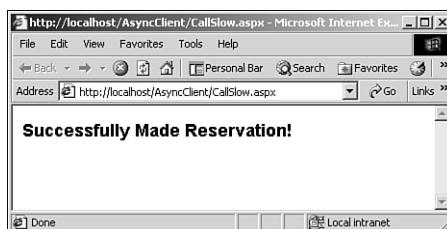
ANALYSIS Lines 17–20 contain the custom callback method for the Web form. This callback varies from the console application example because it stores the asynchronous result object into session state.

Lines 22–31 contain the `Call` method. This method varies only slightly from the console application example because it stores a flag indicating that the Web method has been called.

Lines 33–71 contain the `Page_Load` method. This method contains some complexity because it can be called under a number of different circumstances. First, the method checks to see whether the Web Service has been called (Line 35). If not, it invokes the `Call` method. Next, `Page_Load` checks to see whether the Web method has returned (Line 39). If not, it displays a `Still waiting...` message. The last `else` clause (Lines 43–70) is executed only if the Web method has been called and returned. The method checks to see whether the asynchronous result object has been processed (Line 48). If not, it processes the object and shows a success message (Line 56). If the asynchronous object is processed, Lines 66–68 display a message directing users to another page; this code makes sure that the asynchronous result object isn't processed twice.

The result of a successful call by the asynchronous client is shown in Figure 16.5.

FIGURE 16.5
Asynchronous Web
form client results after
a Web method call has
completed.



Summary

Today's lesson showed how to use session state in a Web Service by giving an example that tracked the amount of time a Web Service spent in a method call for a user. Clients that don't support cookies, such as Windows forms programs and console applications, can have trouble dealing with Web Services that use session state. You saw some techniques for dealing with the issue in the first part of today's lesson.

You learned how to make asynchronous calls to Web Services in the second part of the lesson. This advanced technique can be especially helpful for Web Services that don't return results immediately. You saw how to create client programs that use asynchronous calls from a Web form and from a console application.

Today's lesson is the last one that deals explicitly with Web Services. Future lessons, such as Day 18, "Configuring Internet Applications," will show you how to configure Web Services, and Day 19, "Securing Internet Applications," will include sample programs that allow you to create secure services.

Q&A

Q I plan on using a commercial Web Service for one of my projects, and I don't have any control over the way this service is implemented. Can I still use asynchronous calling techniques, like the ones described in this lesson?

A Yes. Creating an asynchronous Web Service client doesn't depend on the remote Web Service that you are using. Visual Studio .NET and the .NET framework allow you to create asynchronous clients for any kind of Web Service. The key methods in the Web Service proxy class, such as `BeginSomeMethod` and `EndSomeMethod`, are generated for you automatically.

Q I plan on creating a standalone Windows Forms application that will call a Web Service asynchronously. Which technique in this lesson should I use?

A Windows Forms applications are similar to the console application from Listing 16.9, in that they don't support client-side cookies. You can use the techniques from Listing 16.9 in Windows Forms applications to call a Web Service asynchronously. You would need to move the code in the console application's `Main` method into the Form event handler that calls the Web Service.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered today as well as exercises to give you experience using what you have learned. Try to understand the quiz and exercise before continuing to tomorrow's lesson. Answers are provided in Appendix A, "Answers to Quizzes and Exercises."

Quiz

1. What attribute must be used to enable session state for a public Web method?
2. Why does using session state in a Web Service cause complications for some client programs?

3. Does a special attribute enable application state for Web Services?
4. What technique is used to deal with Web Services that take a long time to return a result from a method call?
5. Do you have to take any special actions to enable asynchronous calling for a Web Service?
6. What interface do objects returned from asynchronous method calls support?
7. Do Web forms support asynchronous method calls to Web Services?

16

Exercises

1. Modify `SlowService` to return error messages at random.
2. Modify one of the asynchronous clients to handle the randomly generated error messages from the new `SlowService` created in Exercise 1.

CHAPTER

23

Multi-Threading

IN THIS CHAPTER

- Creating New Threads **498**
- Synchronization **499**

Multi-threading is the capability to execute multiple threads simultaneously. As systems become more sophisticated, it's often desirable to take advantage of machine capabilities such as multi-processor architectures. Even on single processing machines, multi-threading provides performance enhancements in some applications.

Besides boosting performance, multi-threading is handy in other scenarios such as event-driven operating environments. For instance, it's often necessary to invoke a progress indicator in its own thread so it can receive updates to an ongoing operation in another thread. Multi-threaded programs allow a program to engage in simultaneous events that can provide a user with real-time feedback.

Creating New Threads

The act of creating and invoking a new thread in C# is relatively straightforward. The process involves creating a `Thread` object, instantiating a `ThreadStart` delegate with a delegate method handler, and passing the `ThreadStart` delegate to the new `Thread` object. All that's remaining is to start the thread, and off it runs. Listing 23.1 shows how to create and execute a new thread.

LISTING 23.1 Creating a New Thread: SingleThread.cs

```
using System;
using System.Threading;

/// <summary>
///     Shows how to create a single thread of execution.
/// </summary>
class SingleThread
{
    static void Main(string[] args)
    {
        SingleThread st = new SingleThread();

        Thread th = new Thread(new ThreadStart(st.SayHello));

        th.Start();
    }

    public void SayHello()
    {
        Console.WriteLine("Hello from a single thread.");
    }
}
```

And here's the output:

```
Hello from a single thread.
```

In Listing 23.1, an object of type `SingleThread` is instantiated within the `Main()` method. It contains the `SayHello()` method, which is executed as part of the thread in this program. All of the thread creation and initialization occurs in the following line:

```
Thread th = new Thread(new ThreadStart(st.SayHello));
```

The `Thread` object is declared as `th`. It's instantiated as a new `Thread` object with a new `ThreadStart` delegate as its parameter. The delegate method handler for the `ThreadStart` delegate is the `SayHello()` method of the `SingleThread` object, `st`.

Now the thread exists, but it's idle, waiting for directions. It's said to be in the `unstarted` state. To get this thread running, the program invokes the `Start()` method of the `Thread` object, `th`.

Synchronization

Using the techniques from Listing 23.1, it's easy to create multiple threads of execution. As long as each thread minds its own business, the program runs fine. However, in many situations, this is not practical. It's often necessary for multiple threads to share a resource. Without control, the behavior of multi-threaded programs sharing a resource yields non-deterministic results.

To provide that control, C# allocates methods to coordinate activities between threads. This coordination is properly termed *synchronization*. Correct implementation of synchronization enables programs to take advantage of performance benefits of multi-threading as well as maintaining the integrity of object state and data.

This section uses the C# `lock` keyword to provide data synchronization. The code in Listing 23.2 shows how.

LISTING 23.2 Synchronized Data Access: `Synchronization.cs`

```
using System;
using System.Threading;

/// <summary>
///     Synchronized data.
/// </summary>
class SyncData
{
    int index = 0;
```

LISTING 23.2 continued

```
string[] comment = new string[]
{ "one", "two", "three", "four", "five",
  "six", "seven", "eight", "nine", "ten" };

public string GetNextComment()
{
    // allow only a single thread at a time
    lock (this)
    {
        if (index < comment.Length)
        {
            return comment[index++];
        }
        else
        {
            return "empty";
        }
    }
}

/// <summary>
/// Demonstrates synchronized data access.
/// </summary>
class Synchronization
{
    SyncData sdat = new SyncData();

    static void Main(string[] args)
    {
        Synchronization sync = new Synchronization();

        Thread t1 = new Thread(new ThreadStart(sync.GetComments));
        Thread t2 = new Thread(new ThreadStart(sync.GetComments));
        Thread t3 = new Thread(new ThreadStart(sync.GetComments));

        t1.Name = "Thread 1";
        t2.Name = "Thread 2";
        t3.Name = "Thread 3";

        t1.Start();
        t2.Start();
        t3.Start();
    }

    public void GetComments()
    {
        string comment;
```

LISTING 23.2 continued

```
do
{
    comment = sdat.GetNextComment();

    Console.WriteLine(
        "Current Thread: {0}, comment: {1}",
        Thread.CurrentThread.Name, comment);

    } while (comment != "empty");
}
}
```

Here's sample output from Listing 23.2:

```
Current Thread: Thread 1, comment: one
Current Thread: Thread 3, comment: two
Current Thread: Thread 2, comment: three
Current Thread: Thread 1, comment: four
Current Thread: Thread 1, comment: five
Current Thread: Thread 1, comment: six
Current Thread: Thread 1, comment: seven
Current Thread: Thread 1, comment: eight
Current Thread: Thread 1, comment: nine
Current Thread: Thread 1, comment: ten
Current Thread: Thread 1, comment: empty
Current Thread: Thread 3, comment: empty
Current Thread: Thread 2, comment: empty
```

There are three threads of execution in Listing 23.2 that obtain synchronized access to data. The construction of the threads in the `Main()` method of the `Synchronization` class is similar to how threads were created in Listing 23.1. To keep track of each thread, the program sets each thread's `Name` property.

The `GetComments()` method of the `Synchronization` class is run by each thread. This method obtains a new piece of data, `comment`, from a `SyncData` object and prints its value to the console. The loop ends when the `SyncData` object returns the string `empty`.

The `SyncData` object provides synchronized access to its data. The only way to get to the data is through the public `GetNextComment()` method. Within this method is an `if` statement, keeping data reads from going beyond the bounds of the array. Until `index` reaches the end of the array, the next comment is returned and `index` is incremented so the next thread gets the next comment. When `index` reaches the end of the array, the method returns the string `empty` to signify that there is no more data to return.

Surrounding the `if` statement in the `GetNextComment()` method is a `lock` statement. Here's a cutout of the `lock` statement from Listing 23.2:

```
lock (this)
{
    // statements
}
```

The parameter of the `lock` statement is `this`. The parameter for a `lock` statement can be any reference type expression. An invalid expression would be a value type, such as an `int` type. The `lock` statement implements mutual exclusion on the statements inside the curly braces.

Tip

Use the `lock` statement for mutually exclusive access to data in a multi-threaded program.

Without the `lock` statement, it would be possible for two or more threads to be reading the same value at the same time. In the absence of a `lock` statement, if the statements inside the curly braces represented an airline seat reservation or a bank account withdrawal, the results would not be nice. The `lock` statement ensures that only one thread at a time can be executing those statements.

Summary

This chapter presented multi-threaded applications in C#. The first section discussed how to create and start a thread, including declaring a thread argument and passing it a delegate with the method to be invoked, as well as executing the thread.

To keep threads from wreaking havoc with shared data, it's often necessary to use synchronization objects. Proper thread synchronization helps manage access to program data. The example program in this chapter used `lock` statements, providing a mutual exclusion access scenario to program data.

Multi-threading is common on server programs that create new threads to handle client requests. The next chapter, “Browsing the Network Libraries,” shows how to create clients and servers that communicate over a network.

CHAPTER

25

25

String Manipulation

IN THIS CHAPTER

- The `String` Class 516
- The `StringBuilder` Class 533
- String Formatting 540
- Regular Expressions 541

The base class libraries include an extensive set of APIs for working with strings. This chapter goes beyond the `string` type and introduces specialized classes that make working with strings easier.

The `String` class is similar to the `string` type, but with much more power. Although the `String` class is very robust, it is also immutable, which means that once a `String` object is created, it can't be modified.

When there is a need to manipulate strings, use the `StringBuilder` class. The `StringBuilder` class isn't as streamlined as the `String` class, but it is built especially for modifying strings in any way necessary.

A topic related to strings is regular expressions. The `String` and `StringBuilder` classes have many capabilities, but regular expressions beat them both, hands down, at searching and text matching. The following sections go into detail about `String` and `StringBuilder` types, string formatting, and regular expressions.

Tip

The `String` and `StringBuilder` classes contain instance and static methods. When a `String` or `StringBuilder` instance already exists, use the instance method. However, you can gain performance advantages by avoiding unnecessary instantiations and using static methods.

The `String` Class

The `String` class mirrors the functionality of the `string` type, plus much more. There are numerous methods that compare, read, and search a `String` object's contents.

Strings are immutable, meaning that they can't be modified once created. All methods that appear to modify a `String` really don't. They actually return a new `String` that has been modified based on the method invoked. When heavy modification is needed, use the `StringBuilder` class, which is discussed in the next section.

The `String` class is also sealed. This means that it can't be inherited. Being immutable and sealed makes the `String` class more efficient. Now, let's check out what the `String` class has to offer by examining its methods.

static Methods

The `String` class has several `static` methods. These are class methods that don't need an instance of the `String` class to be invoked. The following paragraphs discuss the `static` `String` class methods.

The `Compare()` Method

The `static Compare()` method compares two strings, which are referred to here as `str1` (string one) and `str2` (string two). It produces the following integer results:

- `str1 < str2` = negative
- `str1 == str2` = zero
- `str1 > str2` = positive

An empty string, `" "`, is always greater than null. Here's an example of how to implement the `Compare()` method:

```
int intResult;
string str1 = "string 1";
string str2 = "string 2";

intResult = String.Compare(str1, str2);

Console.WriteLine("String.Compare({0}, {1}) = {2}\n",
    str1, str2, intResult);
```

The `Compare()` method has the following overloads:

- `Compare(str1, str2, ignoreCase)`
- `Compare(str1, str2, ignoreCase, CultureInfo)`
- `Compare(str1, index1, str2, index2, length)`
- `Compare(str1, index1, str2, index2, length, ignoreCase)`
- `Compare(str1, index1, str2, index2, length, ignoreCase, CultureInfo)`

In these overloads, `str1` and `str2` are the strings to be compared, and `index1` and `index2` are the respective integer offsets into those strings to begin making the comparison. The `length` parameter is the number of characters to compare. The `ignoreCase` is a `bool` parameter where `true` means to ignore character case and `false` means to make a case-sensitive comparison. `CultureInfo` is a class for specifying localization information.

The CompareOrdinal() Method

The static `CompareOrdinal()` method compares two strings—`str1` (string one) and `str2` (string two)—independent of localization. It produces the following integer results:

- `str1 < str2` = negative
- `str1 == str2` = zero
- `str1 > str2` = positive

An empty string, "", is always greater than null. Here's an example of how to implement the `CompareOrdinal()` method:

```
int intResult;
string str1 = "string 1";
string str2 = "string 2";

intResult = String.CompareOrdinal(str2, str1);

Console.WriteLine("String.CompareOrdinal({0}, {1}) = {2}\n",
    str2, str1, intResult);
```

The `CompareOrdinal()` method has the following overload:

- `CompareOrdinal(str1, index1, str2, index2, length)`

In this overload, `str1` and `str2` are the strings to be compared, and `index1` and `index2` are the respective integer offsets into those strings to begin making the comparison. The `length` parameter is the number of characters to compare.

The Concat() Method

The static `Concat()` method creates a new string from one or more input strings or objects. Here's an example of how to implement the `Concat()` method using two strings:

```
int intResult;
string str1 = "string 1";
string str2 = "string 2";

stringResult = String.Concat(str1, str2);

Console.WriteLine("String.Concat({0}, {1}) = {2}\n",
    str1, str2, stringResult);
```

The example shows the `Concat()` method accepting two string parameters. The result is a single string with the second string concatenated to the first. The `Concat()` method has the following overloads:

- `Concat (Object)`

- Concat (Object[])
- Concat (String[])
- Concat (Object, Object)
- Concat (Object, Object, Object)
- Concat (string, string, string)
- Concat (string, string, string, string)

In these overloads, all object parameters are converted to `String` objects before concatenation. The elements of the array parameters are concatenated in order to create a single string. The other overloads, with multiple parameters, form a single `String` by concatenating each of the parameters in the order they appear.

The `Copy()` Method

The static `Copy()` method returns a copy of a `String`. Here's an example of how to implement the `Copy()` method:

```
string stringResult;
string str1 = "string 1";

stringResult = String.Copy(str1);

Console.WriteLine("String.Copy({0}) = {1}\n",
    str1, stringResult);
```

The `Copy()` method makes a copy of `str1`. The result is a copy of `str1` placed in `stringResult`.

The `Equals()` Method

The static `Equals()` method determines whether two strings are equal, returning a `bool` value of true when they are equal and a `bool` value of false when they're not. Here's an example of how to implement the `Equals()` method:

```
bool boolResult;
string str1 = "string 1";
string str2 = "string 2";

boolResult = String.Equals(str1, str2);

Console.WriteLine("String.Equals({0}, {1}) = {2}\n",
    str1, str2, boolResult);
```

The `Equals()` method accepts the two string parameters. The result is a `bool` that will evaluate to `false` because `str1` and `str2` are not the same value.

The Format() Method

The static `Format()` method returns a textual representation of an object after applying a specified format string. Here's an example of how to implement the `Format()` method:

```
string stringResult;
string str1 = "string 1";
string str2 = "string 2";

String formatString = "{0,15}";

stringResult = String.Format(formatString, str2);

Console.WriteLine("String.Format({0}, {1}) = [{2}]\n",
    formatString, str2, stringResult);
```

This example shows the `Format()` method accepting two string parameters. The first parameter is a format string that will be applied to the second parameter. The result is a 15-character string with the text right-aligned and padded to the left with spaces. The `Format()` method has the following overloads:

- `Format (string, Object[])`
- `Format (IFormatProvider, string, Object[])`
- `Format (string, Object, Object)`
- `Format (string, Object, Object, Object)`

In these overloads, the `string` parameter specifies the format string. Whether an array or individual object, each `Object` parameter is formatted according to its corresponding placeholder in the format string. The `IFormatProvider` is an interface that is implemented by an object for managing formatting.

The Intern() Method

The static `Intern()` method returns a reference to a string in the string intern pool. A C# program maintains a string intern pool where literal and constant strings are automatically placed. When strings are built on-the-fly (or programmatically), they are separate objects and are not intern pool members. The `Intern()` method will accept a parameter with a string that was programmatically constructed and return a reference to the identical string from the intern pool. Here's an example of how to implement the `Intern()` method:

```
string str1      = "string1";
String objStr1   = String.Concat("string", "1");
String internedStr1 = String.Intern(objStr1);
```

```
Console.WriteLine(  
    "(object)objStr1 == (object)str1 is {0}\n",  
    ((object)objStr1 == (object)str1));  
  
Console.WriteLine(  
    "(object)internedStr1 == (object)str1 is {0}\n",  
    ((object)internedStr1 == (object)str1));
```

The example shows the effects of using the `Intern()` method on a programmatically constructed string. The `Concat()` method constructs a string on-the-fly, `objStr1`, that is identical in value to `str1`. `objStr1` is a separate object and not a member of the intern string pool. The `Intern()` method returns a reference to a value in the intern pool that is identical to the value of `objStr1` (values are the same, but references are still different). The first `WriteLine()` method will return the value `false` because `objStr1` refers to a separate object, and `str1` refers to a literal string that was added to the intern pool. The second `WriteLine()` method returns `true` because `internedStr1` received a reference to the intern pool, which is the same as `str1` (references are the same).

The `IsInterned()` Method

The static `IsInterned()` method returns a reference to an interned string if it is a member of the intern pool. Otherwise, it returns `null`. Here's an example of how to implement the `IsInterned()` method:

```
stringResult = String.IsInterned(internedStr1);  
  
Console.WriteLine("String.IsInterned({0}) = {1}\n",  
    internedStr1, stringResult);
```

The example shows the `IsInterned()` method determining whether a string is in the intern pool. Assuming that the `internedStr1` string parameter has been interned, the `IsInterned()` method will return a reference to that string in the intern pool.

Note

The intern pool is a system table that eliminates duplication by allowing multiple references to the same constant string when the strings are identical. This saves system memory. The intern-related methods of the `String` class enable a program to determine if a string is interned and to place it in the intern pool to take advantage of the associate memory optimizations.

The `Join()` Method

The static `Join()` method concatenates strings with a specified separator between them. Here's an example of how to implement the `Join()` method:

```
string stringResult;
string str1 = "string 1";
string str2 = "string 2";

String[] strArr = new String[] { str1, str2 };

stringResult = String.Join(", ", strArr);

Console.WriteLine(
    "String.Join(\"\", \"", [str1 and str2]) = {0}\n",
    stringResult);
```

This example shows how to create a comma-separated list of strings with the `Join()` method. The first parameter of the `Join()` method specifies the separator character, a comma in this case. The second parameter is an array of strings that will be separated, resulting in a string where each member of the array is separated by the separation character. The `Join()` method has the following overload:

- `Join(string, stringArray, int1, int2)`

In the preceding overload, `string` is the separator character and `stringArray` is an array of strings to be separated. The next two parameters are the beginning array element to start separating (that is, the first array element to be followed by the separation character) and the number of elements in the array to be separated.

Instance Methods

Instance `String` methods act upon an existing `String` object. Often referred to as this `String`, the instance acted upon by these methods is the same instance that is invoking the method.

The `Clone()` Method

The `Clone()` method returns `this String`. Here's an example of how to implement the `Clone()` method:

```
String stringResult;
string str1 = "string 1";

stringResult = (String)str1.Clone();

Console.WriteLine("(String){0}.Clone() = {1}\n",
    str1, stringResult);
```

The example demonstrates how the `Clone()` method returns a reference to the same instance it is invoked upon. Since the `Clone()` method returns an `Object` reference, the return value must be cast to a `String` before assignment to `stringResult`.

The `CompareTo()` Method

The `CompareTo()` method compares the value of `this` instance with a string. It produces the following integer results:

- `this < string` = negative
- `this == string` = zero
- `this > string` = positive
- `string is null` = 1

An empty string, "", is always greater than `null`. If both `this` and `string` are `null`, then they are equal (zero result). Here's an example of how to implement the `CompareTo()` method:

```
int intResult;
string str1 = "string 1";
string str2 = "string 2";

intResult = str1.CompareTo(str2);

Console.WriteLine("{0}.CompareTo({1}) = {2}\n",
    str1, str2, intResult);
```

The `CompareTo()` method has the following overload:

- `CompareTo(Object)`

In this overload, the `Object` parameter must be a `String`.

The `CopyTo()` Method

The `CopyTo()` method copies a specified number of characters from `this String` to an array of characters. Here's an example of how to implement the `CopyTo()` method:

```
string str1 = "string 1";

char[] charArr = new char[str1.Length];
str1.CopyTo(0, charArr, 0, str1.Length);

Console.WriteLine(
    "{0}.CopyTo(0, charArr, 0, str1.Length) = ",
    str1);
```

```
foreach(char character in charArr)
{
    Console.Write("{0} ", character);
}
Console.WriteLine("\n");
```

This example shows the `CopyTo()` method filling a character array. It copies each character from `str1` into `charArr`, beginning at position 0 and continuing for the length of `str1`. The `foreach` loop iterates through each element of `charArr`, printing the results.

The `EndsWith()` Method

The `EndsWith()` method determines if a `String` suffix matches a specified `String`. Here's an example of how to implement the `EndsWith()` method:

```
bool boolResult;
string str1 = "string 1";
string str2 = "string 2";

boolResult = str1.EndsWith("2");

Console.WriteLine("{0}.EndsWith(\"2\") = {1}\n",
    str1, boolResult);
```

In this case, the `EndsWith()` method checks to see if `str1` ends with the number 2. The result is `false` because `str1` ends with the number 1.

The `Equals()` Method

The `static Equals()` method determines whether two strings are equal, returning a `bool` value of `true` when they are equal and a `bool` value of `false` when they're not. Here's an example of how to implement the `Equals()` method:

```
int intResult;
string str1 = "string 1";
string str2 = "string 2";

boolResult = str1.Equals(str2);

Console.WriteLine("{0}.Equals({1}) = {2}\n",
    str1, str2, boolResult);
```

In this example the `Equals()` method accepts one string parameter. Since `str1` has a different value than `str2`, the return value is `false`. The `Equals()` method has a single instance overload:

- `Compare(Object)`

In the overload, the `Object` parameter must be a `String`.

The GetEnumerator() Method

The `GetEnumerator()` method returns a `CharacterEnumerator` for this `String`. The `foreach` statement uses `IEnumerator` to iterate through a collection. The `CharacterEnumerator`, returned by this method, is an `IEnumerator`. Here's an example of how to implement the `GetEnumerator()` method:

```
string str1 = "string 1";  
  
CharEnumerator charEnum = str1.GetEnumerator();  
  
Console.WriteLine("charEnum is IEnumarator: {0}",  
    charEnum is IEnumarator);
```

The `CharacterEnumerator` inherits from the `System.Collections.IEnumerator` interface. This is why the value returned from the `charEnum is IEnumarator` expression in the `Console.WriteLine()` method will be true.

The IndexOf() Method

The `IndexOf()` method returns the position of a string or characters within this `String`. When the character or string is not found, `IndexOf()` returns `-1`. Here's an example of how to implement the `IndexOf()` method:

```
int intResult;  
string str1 = "string1";  
  
intResult = str1.IndexOf('1');  
  
Console.WriteLine("str1.IndexOf('1'): {0}", intResult);
```

The return value of this operation is `6` because that's the zero-based position within `str1` that the character '`1`' occurs. The `IndexOf()` method has the following overloads:

- `IndexOf(char[])`
- `IndexOf(string)`
- `IndexOf(char, beginInt)`
- `IndexOf(char[], beginInt)`
- `IndexOf(string, beginInt)`
- `IndexOf(char, beginInt, endInt)`
- `IndexOf(char[], beginInt, endInt)`
- `IndexOf(string, beginInt, endInt)`

In these overloads, `char[]` parameters return the first instance of any character in the `char` array, and `string` parameters specify that the first instance of that string should be

searched for. The `beginInt` parameter means to start searching at that index within `this String`, and the `endInt` means to stop searching at that position within `this String`.

The `Insert()` Method

The `Insert()` method returns a string where a specified string is placed in a specified position of `this String`. All characters at and to the right of the insertion point are pushed right to make room for the inserted string. Here's an example of how to implement the `Insert()` method:

```
string stringResult;
string str2 = "string2";

stringResult = str2.Insert(6, "1");

Console.WriteLine("str2.Insert(6, \"1\"): {0}",
    stringResult);
```

This example places a "1" into `str2`, producing "string12".

The `LastIndexOf()` Method

The `LastIndexOf()` method returns the position of the last occurrence of a string or characters within `this String`. Here's an example of how to implement the `LastIndexOf()` method:

```
int intResult;
string stateString = "Mississippi";

intResult = stateString.LastIndexOf('s');

Console.WriteLine("stateString.LastIndexOf('s'): {0}",
    intResult);
```

The preceding example shows how to use the `LastIndexOf()` method to find the position of the last occurrence of the letter 's' in `stateString`. The zero-based result is 6. The `LastIndexOf()` method has the following overloads:

- `LastIndexOf(char[])`
- `LastIndexOf(string)`
- `LastIndexOf(char, beginInt)`
- `LastIndexOf(char[], beginInt)`
- `LastIndexOf(string, beginInt)`
- `LastIndexOf(char, beginInt, endInt)`
- `LastIndexOf(char[], beginInt, endInt)`
- `LastIndexOf(string, beginInt, endInt)`

In these overloads, `char[]` parameters return the last instance of any character in the `char` array, and string parameters specify that the last instance of that string should be searched for. The `beginInt` parameter means to start searching at that index within `this String`, and the `endInt` means to stop searching at that position within `this String`.

The PadLeft() Method

The `PadLeft()` method right aligns the characters of a string and pads on the left with spaces (by default) or a specified character. Here's an example of how to implement the `Equals()` method:

```
string stringResult;
string str1 = "string 1";

stringResult = str1.PadLeft(15);

Console.WriteLine("str1.PadLeft(15): [{0}]",
    stringResult);
```

In this example the `PadLeft()` method creates a 15-character string with the original string right aligned and filled to the left with space characters. The `PadLeft()` method has the following overload:

- `PadLeft(int, char)`

It accepts an integer specifying the number of characters for the new string and a `char` parameter for the padding character.

The PadRight() Method

The `PadRight()` method left aligns the characters of a string and pads on the right with spaces (by default) or a specified character. Here's an example of how to implement the `PadRight()` method:

```
string stringResult;
string str1 = "string 1";

stringResult = str1.PadRight(15, '*');

Console.WriteLine("str1.PadRight(15, '*'): [{0}]",
    stringResult);
```

The example shows the `PadRight()` method creating a 15-character string with the original string left aligned and filled to the right with '*' characters. The `PadRight()` method has the following overload:

- `PadRight(int)`

It accepts an integer specifying the number of characters for the new string, and it defaults to a space for the padding character.

The Remove() Method

The `Remove()` method deletes a specified number of characters from a position in `this String`. Here's an example of how to implement the `Remove()` method:

```
string stringResult;
string str2 = "string2";

stringResult = str2.Remove(3, 3);

Console.WriteLine("str2.Remove(3, 3): {0}",
    stringResult);
```

This example shows the `Remove()` method deleting the fourth, fifth, and sixth characters from `str2`. The first parameter is the zero-based starting position to begin deleting, and the second parameter is the number of characters to delete. The result is “str2”, where the “ing” was removed from the original string.

The Replace() Method

The `Replace()` method replaces all occurrences of a character or string with a new character or string, respectively. Here's an example of how to implement the `Replace()` method:

```
int intResult;
string str2 = "string 2";

stringResult = str2.Replace('2', '5');

Console.WriteLine("str2.Replace('2', '5'): {0}",
    stringResult);
```

In this example the `Replace()` method accepts two character parameters. The first parameter is the char to be replaced, and the second parameter is the char that will replace the first. The `Replace()` method has the following overload:

- `Replace(string, string)`

In this overload, all occurrences of the first string are replaced by the second string.

The Split() Method

The `Split()` method extracts individual strings separated by a specified set of characters and places each of those strings into a string array. Here's an example of how to implement the `Split()` method:

```
String csvString = "one, two, three";  
  
string[] stringArray = csvString.Split(new char[] {',','});  
  
foreach( string strItem in stringArray )  
{  
    Console.WriteLine("Item: {0}", strItem);  
}
```

The example shows the `Split()` method extracting strings that are separated by commas. The individual strings “one”, “two”, and “three” are placed into a different index of `stringArray`. Notice the spaces before the strings “two” and “three”; that is how the `Split()` method preserves white space. The `Split()` method has the following overload:

- `Split(char[], int)`

In this overload, `char[]` is an array of characters used as separators, and `int` is the number of strings to place into the resulting array.

The `StartsWith()` Method

The `StartsWith()` method determines if a `String` prefix matches a specified `String`. Here’s an example of how to implement the `StartsWith()` method:

```
bool boolResult;  
string str1 = "string 1";  
  
boolResult = str1.StartsWith("Str");  
  
Console.WriteLine("str1.StartsWith(\"Str\"): {0}",  
    boolResult);
```

In this case, the `StartsWith()` method checks to see if `str1` begins with the “Str”. The result is `false` because `str1` begins with “str”, where the first character is lowercase.

The `Substring()` Method

The `SubString()` method retrieves a substring at a specified location from this `String`. Here’s an example of how to implement the `SubString()` method:

```
string stringResult;  
string str1 = "string1";  
  
stringResult = str1.Substring(3);  
  
Console.WriteLine("str1.Substring(3): {0}",  
    stringResult);
```

The result of this example is “ing1”. The `SubString()` method has the following overload:

- `SubString (int, int)`

The first `int` is the zero-based position to begin extracting the substring from, and the second parameter is the number of characters to get.

The `ToCharArray()` Method

The `ToCharArray()` method copies the characters from `this String` into a character array. Here’s an example of how to implement the `ToCharArray()` method:

```
int intResult;
string str1 = "string1";

char[] characterArray = str1.ToCharArray();

foreach( char character in characterArray )
{
    Console.WriteLine("Char: {0}", character);
}
```

The `ToCharArray()` method has the following overload:

- `ToCharArray(int, int)`

The first `int` specifies the beginning of a substring to copy to the character array, and the second parameter indicates how many characters to move.

The `ToLower()` Method

The `ToLower()` method returns a copy of `this string` converted to lowercase characters. Here’s an example of how to implement the `ToLower()` method:

```
string stringResult;
string ucString = "UpperCaseString";

stringResult = ucString.ToLower();

Console.WriteLine("ucString.ToLower(): {0}",
    stringResult);
```

The result of this example converts “UpperCaseString” to “uppercasestring”. The `ToLower()` method has the following overload:

- `ToLower(CultureInfo)`

`CultureInfo` is a class for specifying localization information.

The `ToUpper()` Method

The `ToUpper()` method returns a copy of `this String` converted to uppercase characters. Here's an example of how to implement the `ToUpper()` method:

```
string stringResult;
string str1 = "string1";

stringResult = str1.ToUpper();

Console.WriteLine("str1.ToUpper(): {0}",
    stringResult);
```

In this example, the result converts “string1” to “STRING1”. The `ToUpper()` method has the following overload:

- `ToUpper(CultureInfo)`

`CultureInfo` is a class for specifying localization information.

The `Trim()` Method

The `Trim()` method removes whitespace or a specified set of characters from the beginning and ending of `this String`. Here's an example of how to implement the `Trim()` method:

```
string stringResult;
string trimString = " nonwhitespace ";

stringResult = trimString.Trim();

Console.WriteLine("trimString.Trim(): [{0}]",
    stringResult);
```

The example shows the `Trim()` method being used to remove all the whitespace from the beginning and end of `trimString`. The result is “nonwhitespace”, with no spaces on either side. The `Trim()` method has the following overload:

- `Trim(char[])`

In this overload, `char[]` is an array of characters that are trimmed from the beginning and end of a string.

The `TrimEnd()` Method

The `TrimEnd()` method removes a specified set of characters from the end of `this String`. Here's an example of how to implement the `TrimEnd()` method:

```
string stringResult;
string trimString = " nonwhitespace ";
```

```
stringResult = trimString.TrimEnd(new char[] {' '});  
  
Console.WriteLine("trimString.TrimEnd(): [{0}]",  
    stringResult);
```

In this example the `TrimEnd()` method removes all the whitespace from the end of `trimString`. The result is “nonwhitespace”, with no spaces on the right side.

The `TrimStart()` Method

The `Trim()` method removes whitespace or a specified set of characters from the beginning of this `String`. Here’s an example of how to implement the `Trim()` method:

```
string stringResult;  
string trimString = " nonwhitespace ";  
  
stringResult = trimString.TrimStart(new char[] {' '});  
  
Console.WriteLine("trimString.TrimStart(): [{0}]",  
    stringResult);
```

Here, the `TrimStart()` method removes all the whitespace from the beginning of `trimString`. The result is “nonwhitespace”, with no spaces on the left side.

Properties and Indexers

The `String` class has a single property, `Length`, and an indexer.

The `Length` Property

The `Length` property returns the number of characters in a `String`. Here’s an example of how to implement the `Length` property:

```
int intResult;  
string str1 = "string1";  
  
intResult = str1.Length;  
  
Console.WriteLine("str1.Length: {0}",  
    intResult);
```

The example shows the `Length` property being used to get the number of characters in `str1`. The result is 7.

The `String` Indexer

The `String` indexer returns a character within the string at a specified location. Here’s an example of how to implement the `String` indexer:

```
char charResult;  
string str1 = "string 1";  
  
charResult = str1[3];  
  
Console.WriteLine("str1[3]: {0}",  
    charResult);
```

In this example, the indexer extracts the third character from a zero-based count on `str1`. The result is the character ‘i’.

The `StringBuilder` Class

For direct manipulation of a string, use the `StringBuilder` class. It’s the best solution when a lot of work needs to be done to change a string. It’s more efficient for manipulation operations because, unlike a `String` object, it doesn’t incur the overhead involved in creating a new object on every method call. The `StringBuilder` class is a member of the `System.Text` namespace.

Tip

A `String` instantiates and returns a new object when its contents are modified. It’s a good idea to consider using a `StringBuilder` for string modifications to avoid the overhead associated with additional instantiations of modified string objects.

Instance Methods

The `StringBuilder` class doesn’t have static methods. All of its methods operate on the instance they’re invoked from. The invoking object is referred to in following sections as `this StringBuilder`.

The `Append()` Method

The `Append()` method adds a typed object to `this StringBuilder`. Here’s an example of how to implement the `Append()` method:

```
StringBuilder myStringBuilder;  
myStringBuilder = new StringBuilder("Original");  
  
myStringBuilder.Append("Appended");  
  
Console.WriteLine(  
    "myStringBuilder.Append(\"Appended\"): {0}",  
    myStringBuilder);
```

This example shows how to append one string to another with the `Append()` method. The result is “OriginalAppended”. The `Append()` method has the following overloads:

- `Append(bool)`
- `Append(byte)`
- `Append(char)`
- `Append(char[])`
- `Append(decimal)`
- `Append(double)`
- `Append(short)`
- `Append(int)`
- `Append(long)`
- `Append(Object)`
- `Append(sbyte)`
- `Append(float)`
- `Append(ushort)`
- `Append(uint)`
- `Append(ulong)`
- `Append(char, int)`
- `Append(char[], int, int)`
- `Append(string, int, int)`

In these overloads, each type is converted to its string representation and appended to this `StringBuilder`. The `Append(char, int)` overload appends a specified number, `int`, of `char` to this `StringBuilder`. In the last two overloads, the first `int` specifies the beginning character of either the `char[]` or `string` to start appending, and the second `int` specifies the number of characters to append.

The `AppendFormat()` Method

The `AppendFormat()` method can replace multiple format specifications with a properly formatted value. Here’s an example of how to implement the `AppendFormat()` method:

```
StringBuilder myStringBuilder;
myStringBuilder = new StringBuilder("Original");

myStringBuilder.AppendFormat("{0,10}", "Appended");

Console.WriteLine()
```

```
"myStringBuilder.AppendFormat(\"{0,10}\", \"Appended\"): {0}",  
myStringBuilder);
```

This example uses the `AppendFormat()` method to format the "Appended" string to 10 characters and then append it to `myStringBuilder`. The result is "Original Appended", with two spaces between words because "Appended" was formatted to 10 characters. The `AppendFormat()` method has the following overloads:

- `AppendFormat(string, Object[])`
- `AppendFormat(IFormatProvider, string, Object)`
- `AppendFormat(string, Object, Object)`
- `AppendFormat(string, Object, Object, Object)`

In these overloads, the `string` parameter is the format specification. The `Object` parameters are the object(s) upon which to apply formatting. The `IFormatProvider` is an interface that is implemented by an object to manage formatting.

The `EnsureCapacity()` Method

The `EnsureCapacity()` method guarantees that a `StringBuilder` will have a specified minimal size. Here's an example of how to implement the `EnsureCapacity()` method:

```
StringBuilder myStringBuilder;  
int capacity;  
myStringBuilder = new StringBuilder();  
  
capacity = myStringBuilder.EnsureCapacity(129);  
  
Console.WriteLine(  
    "myStringBuilder.EnsureCapacity(129): {0}",  
    capacity);
```

The example shows the `EnsureCapacity()` method guaranteeing that `myStringBuilder` will have at least a 129-character capacity. The result will have an actual capacity setting of 256, because, interestingly, the capacity is set to the lowest power of two greater than the specified capacity. Setting the minimum capacity to 258, for instance, results in an actual capacity setting of 512.

The `Equals()` Method

The `Equals()` method compares a given `StringBuilder` to this `StringBuilder`. It returns true when both `StringBuilder`s are equal, and false otherwise. Here's an example of how to implement the `Equals()` method:

```
StringBuilder myStringBuilder;  
StringBuilder anotherStringBuilder;
```

```
bool boolResult;  
myStringBuilder = new StringBuilder("my string builder");  
anotherStringBuilder = new StringBuilder("another string builder");  
  
boolResult = myStringBuilder.Equals(anotherStringBuilder);  
  
Console.WriteLine(  
    "myStringBuilder.Equals(anotherStringBuilder): {0}",  
    boolResult);
```

The `Equals()` method in this example compares two `StringBuilder` objects. Since their values are different, the `Equals()` method returns `false`.

The `Insert()` Method

The `Insert()` method places a specified object into this `StringBuilder` at a specified location. Here's an example of how to implement the `Insert()` method:

```
StringBuilder myStringBuilder;  
myStringBuilder = new StringBuilder("one, three");  
  
myStringBuilder.Insert(3, ", two");  
  
Console.WriteLine(  
    "myStringBuilder.Insert(3, \", two\"): {0}",  
    myStringBuilder);
```

The example shows how to insert a string into a `StringBuilder`. The original string, “one, three” becomes “one, two, three”. The `Insert()` method has the following overloads:

- `Insert(int, bool)`
- `Insert(int, byte)`
- `Insert(int, char)`
- `Insert(int, char[])`
- `Insert(int, decimal)`
- `Insert(int, double)`
- `Insert(int, short)`
- `Insert(int, int)`
- `Insert(int, long)`
- `Insert(int, Object)`
- `Insert(int, sbyte)`
- `Insert(int, float)`
- `Insert(int, ushort)`

- `Insert(int, uint)`
- `Insert(int, ulong)`
- `Insert(int, string, countInt)`
- `Insert(int, char[], beginInt, numberInt)`

In the overloads, each type is converted to its string representation and inserted into this `StringBuilder` at the position specified by `int`. The `countInt` specifies the number of strings to insert at `int` position. The `beginInt` and `numberInt` parameters indicate, respectively, where in `char[]` to begin using characters to insert, and `numberInt` indicates how many items from `char[]` to insert.

The Remove() Method

The `Remove()` method deletes a specified span of characters from this `StringBuilder`.

Here's an example of how to implement the `Remove()` method:

```
StringBuilder myStringBuilder;
myStringBuilder = new StringBuilder("Jane X. Doe");

myStringBuilder.Remove(4, 3);

Console.WriteLine(
    "myStringBuilder.Remove(4, 3): {0}",
    myStringBuilder);
```

As the example shows, the first parameter is the zero-based position to begin removing characters, and the second parameter is the number of characters to remove. Removing three characters transforms “Jane X. Doe” into “Jane Doe”.

The Replace() Method

The `Replace()` method replaces a specified set of characters with another. Here's an example of how to implement the `Replace()` method:

```
StringBuilder myStringBuilder;
myStringBuilder = new StringBuilder("Jane X. Doe");

myStringBuilder.Replace('X', 'B');

Console.WriteLine(
    "myStringBuilder.Replace('X', 'B'): {0}",
    myStringBuilder);
```

This example shows the `Replace()` method accepting two string parameters. The first is the character to be replaced, and the second parameter is the character that will replace the first. The result is that “Jane X. Doe” is transformed to “Jane B. Doe”. The `Replace()` method has the following overloads:

- Replace(string, string)
- Replace(char, char, beginInt, numberInt)
- Replace(string, string, beginInt, numberInt)

In these overloads, a second `string` parameter will replace the first, and a second `char` parameter will replace the first `char` parameter. The `beginInt` parameter references the position in `this StringBuilder` to begin replacement, and the `numberInt` indicates the offset from `beginInt` of where to stop replacing.

The `ToString()` Method

The `ToString()` method converts `this StringBuilder` to a string. Here's an example of how to implement the `ToString()` method:

```
StringBuilder myStringBuilder;
string stringResult;
myStringBuilder = new StringBuilder("my string");

stringResult = myStringBuilder.ToString();

Console.WriteLine(
    "myStringBuilder.ToString(): {0}",
    stringResult);
```

The `ToString()` method has the following overload:

- `ToString(beginInt, numberInt)`

In the overload, `beginInt` is the starting position in `this StringBuilder` to start extracting characters, and `numberInt` is the number of characters to convert.

Properties and Indexers

The `String` class has a few properties and an indexer.

The Capacity Property

The `Capacity` property sets and returns the number of characters `this StringBuilder` can hold. Here's an example of how to implement the `Capacity` property:

```
StringBuilder myStringBuilder;
int intResult;
myStringBuilder = new StringBuilder("my string");

intResult = myStringBuilder.Capacity;
```

```
Console.WriteLine(  
    "myStringBuilder.Capacity: {0}",  
    intResult);
```

The result of this example is 16.

The Length Property

The `StringBuilder` `Length` property returns the number of characters in a `String`.

Here's an example of how to implement the `Length` property:

```
StringBuilder myStringBuilder;  
int intResult;  
myStringBuilder = new StringBuilder("my string");  
  
intResult = myStringBuilder.Length;  
  
Console.WriteLine(  
    "myStringBuilder.Length: {0}",  
    intResult);
```

The result of this example is 9.

The MaxCapacity Property

The `MaxCapacity` property is a read-only property that returns the maximum number of characters this `StringBuilder` can hold. Here's an example of how to implement the `MaxCapacity` property:

```
StringBuilder myStringBuilder;  
int intResult;  
myStringBuilder = new StringBuilder("my string");  
  
intResult = myStringBuilder.MaxCapacity;  
  
Console.WriteLine(  
    "myStringBuilder.MaxCapacity: {0}",  
    intResult);
```

The result of this example is 2147483647.

The Indexer

The indexer permits reading and writing of a specified character at a certain position.

Here's an example of how to implement the `Indexer` property:

```
StringBuilder myStringBuilder;  
char charResult;  
myStringBuilder = new StringBuilder("my string");  
  
charResult = myStringBuilder[1];
```

```
Console.WriteLine(
    "myStringBuilder[1]: {0}",
    charResult);
```

The return value in this case is 'y'.

String Formatting

When performing a `Console.WriteLine()` method call, any format strings default to string type unless special formatting is applied. Often it's necessary to perform more sophisticated formatting on various types such as numbers, strings, and dates.

Numeric Formatting

C# has several formatting characters for numeric formatting. Table 25.1 shows the C# number format specifiers.

TABLE 25.1 Numeric Format Specifiers

<i>Format Character</i>	<i>Description</i>
C or c	Currency
D or d	Decimal
E or e	Scientific/exponential
F or f	Fixed point
G or g	General (can be E or F format)
N or n	Number
R or r	Roundtrip (convertible to string and back)
X or x	Hexadecimal

The following example shows how to use numeric formatting:

```
Console.WriteLine("Hex: {0:x}", 255);
```

This example converts the integer 255 to hexadecimal notation. The result is `hex ff`. The value is in lowercase because a lowercase format character was used. Results are in uppercase when uppercase format characters are used.

Picture Formatting

It's often necessary to have more control over the format of output beyond default formatting or a simple numeric formatting character. In these cases, picture formatting will help present output exactly as desired. Table 25.2 shows the picture formatting characters.

TABLE 25.2 Picture Format Characters

Format Character	Description
0	Zero placeholder
#	Display digit placeholder
.	Decimal point
,	Group separator/multiplier
%	Percent
E+0, E-0, e+0, e-0	Exponent notation
\	Literal character
'ABC' or "ABC"	Literal string
;	Section separator

The following example shows how to use picture formatting:

```
Console.WriteLine("Million: {0:$#,#.00}", 1000000);
```

This example formats the number to make it appear as currency. (The C number format character could have been used, but it wouldn't have served the purpose of this example.) The result of this formatting produces “\$1,000,000.00”. The \$ sign is placed into the output at the position it appears. The # symbol holds a place for a number before and after the comma. The ', ' character causes a comma to be placed between every three digits of the output. The decimal point will be placed to the right of the whole number. To get the cents portion to appear, two zeros are put after the decimal point in the format specifier. Had these been # symbols, nothing would have appeared after the decimal point.

Regular Expressions

Regular expressions provide the capability to manipulate and search text efficiently. The `System.Text.RegularExpressions` namespace contains a set of classes that enable regular expression operations in C# programs. Listing 25.1 shows the code for a program similar to grep (Global Regular Expression Print) expressions:

LISTING 25.1 Regular Expressions

```
using System;
using System.Text.RegularExpressions;
using System.IO;

class lrep
{
    static int Main(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Wrong number of args!");
            return 1;
        }

        Regex re = new Regex(args[0]);

        StreamReader sr = new StreamReader(args[1]);

        string nextLine = sr.ReadLine();

        while (nextLine != null)
        {
            Match myMatch = re.Match(nextLine);

            if (myMatch.Success)
            {
                Console.WriteLine("{0}: {1}", args[1], nextLine);
            }

            nextLine = sr.ReadLine();
        }
        sr.Close();
    }
}
```

Note

Global Regular Expression Print (grep), written by Doug McIlroy, is a popular Unix utility. It allows you to perform a command line search for regular expressions within the text of one or more files.

The Listing 25.1 program is called `lrep`, which stands for Limited Regular Expression Print. It may be limited in features, but because of the built-in regular expression classes, it's very powerful. Here's an example of how to use it:

```
lrep string lrep.cs
```

The first parameter, `lrep`, is the command name of the program. The second parameter, `string`, is the regular expression. It happens to be a normal string without anything special, but can also take the same set of regular expressions as the Perl programming language. The third parameter, `lrep.cs`, is the filename to search for the regular expression. Here's the output:

```
lrep.cs: static int Main(string[] args)
lrep.cs:     string nextLine = sr.ReadLine();
```

Each line of output contains the name of the file that was searched. Following that is the text of the line where the regular expression matched. The next example shows how the regular expression is set in the program:

```
Regex re = new Regex(args[0]);
```

A regular expression is created by instantiating a `Regex` object. The following example shows one way to use a regular expression object:

```
Match myMatch = re.Match(nextLine);
```

The `Match()` method of the `Regex` class is used to determine if a given string contains text that matches a regular expression. This program opens a file, specified in the command line arguments, and reads each line to see if there is a match. By using the `Success` property of the match object, the program can figure out that a match was made. This program writes the positive matching lines to the console, as shown in the output lines.

This program used a late bound matching scheme to achieve its goals. However, a regular expression may be initialized with constants, increasing efficiency through compile time optimization.

Summary

There is a plethora of options available in the way of string manipulation with the system libraries. The `String` class provides basic string handling but has many methods available for returning new strings with various modifications.

For sophisticated string manipulation, use the `StringBuilder` class. It allows modification of the string in the same object without the overhead of creating a new object with each operation.

Strings need to be formatted for many processing activities. There are simple number formatting options as well as picture formatting.

A welcome feature of the system libraries is regular expressions. Regular expressions allow powerful string manipulation that is more efficient than either `String` or `StringBuilder` class operations.

Strings and `StringBuilder`s have various features that make them work well as collection objects. The next chapter, “C# Collections,” explains why this is true and provides insight to help understand the internal mechanism of collections.

CHAPTER

28

28

Reflection

IN THIS CHAPTER

- Discovering Program Information **582**
- Dynamically Activating Code **588**
- `Reflection.Emit` **590**

Reflection is the capability to inspect the metadata of a program and gather information about its types. Using reflection, it's possible to learn about program assemblies, modules, and all types of internal program elements.

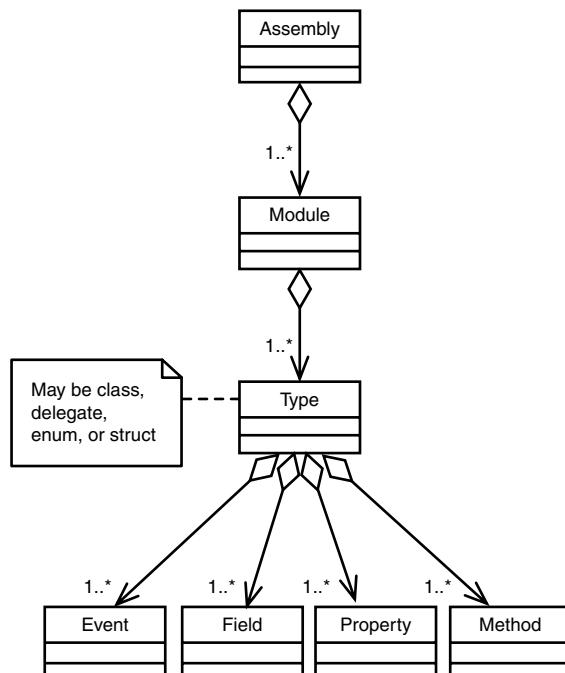
This is particularly useful for design tools, supporting the automated building of code based on user selections derived from the metadata of the underlying types being used. Reflection also provides excellent support for late-bound frameworks where runtime determination is required for selecting required libraries or other functionality on-the-fly.

Discovering Program Information

The reflection API is based on a hierarchical model where higher-level items are composed of one or more lower-level items. Figure 28.1 shows this hierarchy. Assemblies, the basic unit of program distribution in C#, are at the top. Assemblies may be composed of one or more modules. Modules may have one or more types. Types are program elements such as classes, structs, delegates, and enums. Types may contain one or more fields, properties, methods, or events, depending on the type. It may be handy to think about this model as you progress through the sections of this chapter.

FIGURE 28.1

Reflection API hierarchy.



The primary purpose of reflection is to discover program information. The C# reflection API makes it possible to find out all information available about a program. The program elements to be searched include assemblies, modules, types, and type members. Listing 28.1 creates a class that will be reflected upon, and Listing 28.2 demonstrates how to obtain the various reflection objects and inspect their available information.

LISTING 28.1 Class to Reflect Upon: Reflected.cs

```
using System;
using System.Collections;

/// <summary>
///     Reflected Class
/// </summary>
public class Reflected
{
    public int MyField;
    protected ArrayList myArray;

    public Reflected()
    {
        myArray = new ArrayList();
        myArray.Add("Some ArrayList Entry");
    }

    public float MyProperty
    {
        get
        {
            return MyEvent();
        }
    }

    public object this[int index]
    {
        get
        {
            if (index <= index)
            {
                return myArray[index];
            }
            else
            {
                return null;
            }
        }
        set
        {
```

LISTING 28.1 continued

```
        myArray.Add(value);
    }
}

public float MyInstanceMethod()
{
    Console.WriteLine("Invoking Instance MyMethod.");
    return 0.02f;
}

public static float MyStaticMethod()
{
    Console.WriteLine("Invoking Static MyMethod.");
    return 0.02f;
}

public delegate float MyDelegate();

public event MyDelegate MyEvent
    = new MyDelegate(MyStaticMethod);

public enum MyEnum { valOne, valTwo, valThree };
}
```

LISTING 28.2 Performing Reflection: Reflecting.cs

```
using System;
using System.Reflection;

/// <summary>
///     Performing Reflection.
/// </summary>
class Reflecting
{
    static void Main(string[] args)
    {
        Reflecting reflect = new Reflecting();

        Assembly myAssembly
            = Assembly.LoadFrom("Reflecting.exe");

        reflect.GetReflectionInfo(myAssembly);
    }

    void GetReflectionInfo(Assembly myAssembly)
    {
        Type[] typeArr = myAssembly.GetTypes();
```

LISTING 28.2 continued

```
foreach (Type type in typeArr)
{
    Console.WriteLine("\nType: {0}\n", type.FullName);

    ConstructorInfo[] MyConstructors
        = type.GetConstructors();
    foreach (ConstructorInfo constructor
            in MyConstructors)
    {
        Console.WriteLine("\tConstructor: {0}",
                          constructor.ToString());
    }
    Console.WriteLine();

    FieldInfo[] MyFields = type.GetFields();
    foreach (FieldInfo field in MyFields)
    {
        Console.WriteLine("\tField: {0}",
                          field.ToString());
    }
    Console.WriteLine();

    MethodInfo[] MyMethods = type.GetMethods();
    foreach (MethodInfo method in MyMethods)
    {
        Console.WriteLine("\tMethod: {0}",
                          method.ToString());
    }
    Console.WriteLine();

    PropertyInfo[] MyProperties
        = type.GetProperties();

    foreach ( PropertyInfo property in MyProperties)
    {
        Console.WriteLine("\tProperty: {0}",
                          property.ToString());
    }
    Console.WriteLine();

    EventInfo[] MyEvents = type.GetEvents();
    foreach (EventInfo anEvent in MyEvents)
    {
        Console.WriteLine("\tEvent: {0}",
                          anEvent.ToString());
    }
    Console.WriteLine();
}
```

And the output is:

Type: Reflecting

```
Constructor: Void .ctor()

Method: Int32 GetHashCode()
Method: Boolean Equals(System.Object)
Method: System.String ToString()
Method: System.Type GetType()
```

Type: Reflected

```
Constructor: Void .ctor()

Field: Int32 MyField

Method: Int32 GetHashCode()
Method: Boolean Equals(System.Object)
Method: System.String ToString()
Method: Single get_MyProperty()
Method: System.Object get_Item(Int32)
Method: Void set_Item(Int32, System.Object)
Method: Single MyInstanceMethod()
Method: Single MyStaticMethod()
Method: Void add_MyEvent(MyDelegate)
Method: Void remove_MyEvent(MyDelegate)
Method: System.Type GetType()

Property: Single MyProperty
Property: System.Object Item [Int32]

Event: MyDelegate MyEvent
```

Type: Reflected+MyDelegate

```
Constructor: Void .ctor(System.Object, UIntPtr)

Method: Single EndInvoke(System.IAsyncResult)
Method: System.IAsyncResult BeginInvoke(
                    System.AsyncCallback,
                    System.Object)
Method: Single Invoke()
Method: Void GetObjectData()
```

```
System.Runtime.Serialization.SerializationInfo,
System.Runtime.Serialization.StreamingContext)
Method: System.Object Clone()
Method: System.Delegate[] GetInvocationList()
Method: Int32 GetHashCode()
Method: Boolean Equals(System.Object)
Method: System.String ToString()
Method: System.Object DynamicInvoke(System.Object[])
Method: System.Reflection.MethodInfo get_MethodInfo()
Method: System.Object get_Target()
Method: System.Type GetType()

Property: System.Reflection.MethodInfo Method
Property: System.Object Target
```

Type: Reflected+MyEnum

```
Field: Int32 value_
Field: MyEnum valOne
Field: MyEnum valTwo
Field: MyEnum valThree

Method: System.String ToString(System.IFormatProvider)
Method: System.TypeCode GetTypeCode()
Method: System.String ToString(System.String,
                               System.IFormatProvider)
Method: Int32 CompareTo(System.Object)
Method: Int32 GetHashCode()
Method: Boolean Equals(System.Object)
Method: System.String ToString()
Method: System.String ToString(System.String)
Method: System.Type GetType()
```

28

REFLECTION

The primary purpose of Listing 28.1 is to have a class available with all types of class members to reflect upon. It does nothing more than serve that purpose, and all of its program elements should be familiar by now.

Listing 28.2 is where the interesting bits are. The `Main()` method obtains an assembly object by calling the static `LoadFrom()` method of the `Assembly` class. The `LoadFrom()` method has a `string` parameter, specifying the name of the executable file, or assembly, to load.

Within the `GetReflectionInfo()` method, the `Assembly` object, `myAssembly`, invokes its `GetTypes()` method to obtain an array of all types available in the assembly.

Tip

The Assembly type has a `GetModules()` method that will get an array of modules within an assembly. From each of the modules, it's possible to use `GetTypes()` to get an array of types to work with. As a shortcut, Listing 28.2 uses the `GetTypes()` method of the Assembly type to get all types belonging to all modules within that assembly.

Within the `foreach` loop, each type is extracted and printed. The types are obtained with a `Get<X>()` method, and the result is an `<X>Info` object where `<X>` is one of the following type members:

- Constructor
- Field
- Method
- Property (including indexer)
- Event

Each type member is printed to the console with the `ToString()` method, but this isn't the only thing that can be done with each member. Each `<X>Info` class includes numerous methods and properties that can be invoked to obtain information. A good source of information on available methods and properties is the .NET SDK Frameworks documentation. Listing 28.3 shows how to compile the programs in Listings 28.1 and 28.2.

LISTING 28.3 Compile Instructions for Listings 28.1 and 28.2

```
csc Reflecting.cs Reflected.cs
```

Dynamically Activating Code

Dynamic code activation is the capability to make a runtime determination of what code will be executed. This capability can be useful in any situation where a late-bound framework is required.

Consider the Simple Object Access Protocol (SOAP) specification, which is transport protocol independent. Although SOAP is widely used with the HTTP protocol, the specification itself was constructed to allow implementation over other protocols, such as Simple Message Transport Protocol (SMTP). With an appropriate interface, Dynamic Link Libraries (DLL) could be constructed to separate the SOAP implementation from

the underlying protocol. Furthermore, with late-bound implementation, new protocols with the proper interface, packaged in their own DLLs, could be added to the framework at any time, without recompilation of the code. The late-bound capabilities of reflection could enable this scenario by assisting in the runtime determination of what transport protocol would be used for SOAP packages.

Examples in this chapter do not attempt to be this ambitious. However, Listing 28.4 shows how to perform a late-bound operation by dynamically activating the code in a specified assembly during runtime.

LISTING 28.4 Dynamically Activating Code: Reflecting.cs

```
using System;
using System.Reflection;

/// <summary>
///     Dynamically Activating Code.
/// </summary>
class Reflecting
{
    static void Main(string[] args)
    {
        Reflecting reflect = new Reflecting();

        Assembly myAssembly
            = Assembly.LoadFrom("Reflecting.exe");

        reflect.DynamicallyInvokeMembers(myAssembly);
    }

    void DynamicallyInvokeMembers(Assembly myAssembly)
    {
        Type classType = myAssembly.GetType("Reflected");

        PropertyInfo myProperty
            = classTypeGetProperty("MyProperty");

        MethodInfo propGet = myProperty.GetGetMethod();

        object reflectedObject
            = Activator.CreateInstance(classType);

        propGet.Invoke(reflectedObject, null);

        MethodInfo myMethod
            = classTypeGetMethod("MyInstanceMethod");

        myMethod.Invoke(reflectedObject, null);
    }
}
```

And the output is:

```
Invoking Static MyMethod.  
Invoking Instance MyMethod.
```

The `Main()` method of Listing 28.4 gets an `Assembly` object with the static `Assembly.LoadFrom()` method. The `DynamicallyInvokeMembers()` method uses the `Assembly` object to get the `Type` object from the `Reflected` class. The `Type` object is then used to obtain the `MyProperty` property. Next, a `MethodInfo` object is obtained by calling the `GetGetMethod()` of the `PropertyInfo` object. The `GetGetMethod()` retrieves a copy of a property's get method, which is, for reflection purposes, treated just like a method.

Note

Indexer get and set accessors are obtained just like property get and set accessors, with `GetGetMethod()` and `GetSetMethod()` calls.

The `Reflected` class is instantiated by using the `Activator.CreateInstance()` method. The instantiated object is then used as the first parameter in the `Invoke()` method of the `MethodInfo` object. This identifies which object to invoke the method on. The `Invoke()` method's second parameter is the parameter list to send to the method, which would be an array of objects if there were parameters. In this case there are no parameters to send to the method, so the `Invoke()` method's second parameter is set to `null`.

The next two lines show how to dynamically invoke an instance method. The syntax is the same as just explained for the property get accessor. However, the intermediate step, used in properties, isn't necessary, and the method can be obtained directly with the `GetMethod()` method of the `Type` object.

The code in Listing 28.4 can be combined with Listing 28.1 to create an executable. Listing 28.5 shows how to compile them.

LISTING 28.5 Compile Instructions for Listings 28.1 and 28.4

```
csc Reflecting.cs Reflected.cs
```

Reflection.Emit

The `Reflection.Emit` API provides a means to dynamically create new assemblies. Using customized builders and generating Microsoft Intermediate Language (MSIL) or Common Intermediate Language (CIL) code enables programs to create new programs at

runtime. These assemblies may be dynamically invoked or saved to file where they may be reloaded and invoked or used by other programs.

Dynamic assembly creation can be useful for back-ends to compilers or scripting engines on tools such as Web browsers. Using the `Reflection.Emit` API, any tool can be extended to dynamically support .NET or any other Common Language Infrastructure (CLI) compliant system. Listing 28.6 shows how to both generate a dynamic assembly and save it as a console program.

LISTING 28.6 Dynamic Assembly Generation

```
using System;
using System.Reflection;
using System.Reflection.Emit;

/// <summary>
///     Reflection Emit.
/// </summary>
class Emit
{
    static void Main(string[] args)
    {
        AppDomain myAppDomain = AppDomain.CurrentDomain;

        AssemblyName myAssemblyName = new AssemblyName();
        myAssemblyName.Name = "DynamicAssembly";

        AssemblyBuilder myAssemblyBuilder =
            myAppDomain.DefineDynamicAssembly(
                myAssemblyName,
                AssemblyBuilderAccess.RunAndSave);

        ModuleBuilder myModuleBuilder =
            myAssemblyBuilder.DefineDynamicModule(
                "DynamicModule",
                "emitter.netmodule");

        TypeBuilder myTypeBuilder =
            myModuleBuilder.DefineType(
                "EmitTestClass");

        MethodBuilder myMethodBuilder =
            myTypeBuilder.DefineMethod(
                "Main",
                MethodAttributes.Public | MethodAttributes.Static,
                null,
                null);
```

LISTING 28.6 continued

```
ILGenerator myILGenerator
    = myMethodBuilder.GetILGenerator();
myILGenerator.EmitWriteLine(
    "\n\tI must emit, reflection is pretty cool!\n");
myILGenerator.Emit(OpCodes.Ret);

Type myType = myTypeBuilder.CreateType();
object myObjectInstance
    = Activator.CreateInstance(myType);

Console.WriteLine("\nDynamic Invocation:");

MethodInfo myMethod = myType.GetMethod("Main");
myMethod.Invoke(myObjectInstance, null);

myAssemblyBuilder.SetEntryPoint(myMethod);
myAssemblyBuilder.Save("emitter.exe");
}
}
```

Before walking through the code in Listing 28.6, you may want to refer to Figure 28.1, which shows a model of the relationships between Reflection API components. The figure may make it clearer as to why each step is necessary.

New assemblies must be created in a specific AppDomain. Invocation of members belonging to a Type within an assembly must be done in the current AppDomain. Therefore, when this program begins, it gets a new AppDomain object by calling the CurrentDomain() method of the AppDomain class.

To create the entire assembly in Listing 28.6, several steps are required:

1. Create an AssemblyBuilder.
2. Create a ModuleBuilder.
3. Create a TypeBuilder.
4. Create a MethodBuilder.
5. Generate IL.
6. Invoke members or persist assembly.

Each builder is created using a defining method of its parent in the hierarchy. This is another reason why the AppDomain object is required, to get an AssemblyBuilder.

The `AssemblyBuilder` object is created by calling the `DefineDynamicAssembly()` method of the `AppDomain` object. The parameters passed to `DefineDynamicAssembly()` are an `AssemblyName` object and an `AssemblyBuilderAccess` enum. The `AssemblyBuilderAccess` enum has three members: `Run`, `RunAndSave`, and `Save`. `Run` means the assembly can only be invoked in memory; `Save` means that the assembly can only be persisted (saved) to file; and `RunAndSave` means both `Run` and `Save`.

With an `AssemblyBuilder` object, a `ModuleBuilder` object is created. The parameters of the `DefineDynamicModule()` method are a `string` with name for the module and another `string` with the filename the module will be saved as. The example shows that the module filename will be “`emitter.netmodule`”.

Tip

The `DefineDynamicModule()` method has four overloads: two are for run-only modules and the other two are for run and persist modules. To guarantee that a module is included during persistence of an assembly, ensure one of the overloads with the `filename` parameter of the `DefineDynamicAssembly()` method is used.

`TypeBuilder` objects are created with the `DefineType()` method of the `ModuleBuilder` object. The `DefineType()` method takes a single `string` parameter with the name of the `Type`.

The final builder object in Listing 28.6 is the `MethodBuilder`, which is created using the `DefineMethod()` method of the `TypeBuilder` object. `DefineMethod()` has four parameters: `name`, `method attributes`, `return type`, and `parameter types`.

The `name` parameter is a `string` with the name of the method. In this case, it’s the `Main()` method. Since a `Main()` method must be defined as `public` and `static`, the second parameter uses the `Public` and `Static` members of the `MethodAttributes` enum. The `return type` is `null`, which defaults to `void`, and the `parameter types` is also `null` which means that this method does not accept arguments. When a method accepts arguments, the fourth parameter would be an array with the type definitions of each method parameter.

Next, the code is generated. To accomplish this, invoke the `MethodBuilder` object’s `GetILGenerator()` method. This results in an `ILGenerator` class that is used to create code.

This is a very simple method that writes a line of text to the console and returns. The `EmitWriteLine()` and `Emit()` methods perform this task.

Prior to invoking code, a type instance is created by calling the `GetType()` method of the `TypeBuilder` object. The resulting `Type` object is then instantiated with the static `Activator.CreateInstance()` method.

Once an object instance is available, the program gets a `MethodInfo` object and dynamically invokes the method, just like in the last section.

What's really cool about this entire procedure is that you can save the work that was done to a file. With the `Assembly` object, the `SetEntryPoint()` method is invoked with the `MethodInfo` parameter for the dynamically generated `Main()` method. Then the file is saved with the `Save` command, which accepts a single `string` parameter specifying the assembly file name.

Warning

One of the goals of Listing 28.6 was to create an executable console application. For a C# program to run standalone, it must have a `Main()` method. Since the program did have a `Main()` method, it would be easy to assume that everything was good to go. However, the `SetEntryPoint()` method of the `AssemblyBuilder` must still be called or else the program will not run standalone. Remember, the system libraries are cross-language compatible, and you shouldn't make the assumption that they know C#.

The `Save()` method of the `AssemblyBuilder` object creates two files. One file is the module named `emitter.netmodule`. This file can be compiled with other modules to create an executable. The other file is the executable named `emitter.exe`. This is a stand-alone program that will execute when invoked from the command line.

Summary

Reflection provides the capability to discover information about a program at runtime. Pertinent program items that can be reflected upon include assemblies, modules, types, and other kinds of C# program elements.

Another feature of reflection is the capability to dynamically activate code at runtime. This is especially relevant to situations where late-bound operations are required. With reflection, any type of C# code can be loaded and invoked dynamically.

The `Reflection.Emit` API provides advanced features for dynamically creating assemblies. This feature could be used in tools such as scripting engines and compilers. Once the code is created, it can be dynamically invoked or saved to file for later use.

CHAPTER

31

Runtime Debugging

IN THIS CHAPTER

- Simple Debugging **636**
- Conditional Debugging **638**
- Runtime Tracing **641**
- Making Assertions **643**

There are several situations where runtime debugging and tracing are desirable. Often it's easy to turn on debugging in a program, let it run, and watch a console screen for specific printouts representing the state of the program during execution. This is a quick way of isolating system failures during development.

For critical code, it may be useful to install a runtime trace facility. This provides a means to capture real-time information on production code and interact with administrators or analysts on what could be causing a problem.

The system libraries have facilities for supporting runtime debugging and tracing. This includes attributes and switches for conditional debugging and multilevel conditions for controlling trace output. It's also possible to monitor the logical implementation of code with assertions.

The `System.Diagnostics` namespace has two primary classes for runtime debugging: `Debug` and `Trace`. For the most part, their functionality is similar; the primary difference between the two comes from how they are used. The `Debug` class is strictly for development environments and requires a `DEBUG` directive or command-line option to be specified to activate its functionality. The `Trace` class is automatically activated and doesn't require any directive or command-line options. This is because the `Trace` class is for programs to be deployed with debugging capability. Debugging code introduces overhead in a program. If programs should not be deployed with debugging information, which reduces overhead, use the `Debug` class. However, if there's a need to have debugging information available in deployment and the overhead is acceptable, the `Trace` class does the trick.

Simple Debugging

In its simplest form, runtime debugging is just a matter of printing out statements to the console. The `Debug` class, a member of the `System.Diagnostics` namespace, has two methods for supporting explicit debugging: `Write()` and `WriteLine()`. These methods work similar to their `Console` class counterparts. Listing 31.1 shows an example that uses the `WriteLine()` method of the `Debug` class.

LISTING 31.1 A Simple Debugging Example: PlainDebugDemo.cs

```
#define DEBUG

using System;
using System.Diagnostics;

/// <summary>
///     Plain Debug Demo.
```

LISTING 31.1 continued

```
/// </summary>
class PlainDebugDemo
{
    static void DebuggedMethod()
    {
        Debug.WriteLine("Debug: Entered MyMethod()");
    }

    static void Main(string[] args)
    {
        TextWriterTraceListener myListener =
            new TextWriterTraceListener(Console.Out);

        Debug.Listeners.Add(myListener);

        DebuggedMethod();
    }
}
```

And here's the output:

```
Debug: Entered MyMethod()
```

Setting up a program for debugging requires statements to specify where debug output should be sent. The `Main()` method in Listing 31.1 creates a `TextWriterTraceListener` class that directs debugging output to the console window. It then adds the listener to the collection of `Debug` listeners.

Listing 31.1 used a `TextWriter` object, `Console.out`, as its output destination. However, debug output could have been just as well sent to a file by instantiating a `Stream` object and providing it as the parameter to the `TextWriterTraceListener` instantiation. The `TextWriterTraceListener` class also has methods to flush and close debug output with the `Flush()` and `Close()` methods, respectively.

The `Listeners` collection of the `Debug` class accepts any derived `TraceListener` class. Therefore, it's possible to create customized trace listeners by deriving them from either the `TraceListener` or `TextWriterTraceListener` classes.

Once an output destination is set up, the program invokes the `DebuggedMethod()` method, which calls the `WriteLine()` method of the `Debug` class. This produces the output shown following the listing.

There are a couple ways to enable debugging. At the top of Listing 31.1 is a `#define DEBUG` directive, enabling the operation of the `Debug` class. Additionally, Listing 31.2 shows how to enable debugging with the command line option, `/d:DEBUG`. One or the

other of these methods, directive or compilation option, enables debugging, but they both are not required together. If neither of these, directive or compilation option, are present, the `Debug` class does not operate, and there would be no output.

LISTING 31.2 Compilation Instructions for Listing 31.1

```
csc /d:DEBUG PlainDebugDemo.cs
```

Conditional Debugging

A program's capability to turn debugging on and off as needed is called *conditional debugging*. During development, output from debugging can clutter up normal output or force paths of execution that isn't necessary on every run. The `System.Diagnostics` namespace has both attributes and switches to turn debugging on and off as necessary. Listing 31.3 shows how to use attributes to control conditional debugging.

LISTING 31.3 Debugging with Conditional Attributes: ConditionalDebugDemo.cs

```
#define DEBUG

using System;
using System.Diagnostics;

/// <summary>
///     Conditional Debug Demo.
/// </summary>
class ConditionalDebugDemo
{
    static bool Debugging = true;

    [Conditional("DEBUG")]
    static void SetupDebugListener()
    {
        TextWriterTraceListener myListener =
            new TextWriterTraceListener(Console.Out);

        Debug.Listeners.Add(myListener);
    }

    [Conditional("DEBUG")]
    static void CheckState()
    {
        Debug.WriteLineIf(Debugging, "Debug: Entered CheckState()");
    }
}
```

LISTING 31.3 continued

```
static void Main(string[] args)
{
    SetupDebugListener();

    CheckState();
}
```

And here's the output:

```
Debug: Entered CheckState()
```

Two features of Listing 31.3 are of primary interest: the `Conditional` attribute and a Boolean condition on output. The `Conditional` attribute is placed at the beginning of a method that can be turned on and off at will. The condition causing the method to be invoked is either the `#define DEBUG` directive at the top of the listing or the command line `/d:DEBUG` option, shown in Listing 31.4. If neither of these, directive or command line option, is present, the methods with the `Conditional` attribute are invoked when called by the `Main()` method.

LISTING 31.4 Compilation Instructions for Listing 31.3

```
csc /d:DEBUG ConditionalDebugDemo.cs
```

The second item of interest in Listing 31.3 is the Boolean condition parameter of the `WriteLineIf()` method in the `CheckState()` method. The `WriteLineIf()` method of the `Debug` class has a first parameter that takes a `bool`. In the example, the static class field `Debugging` is used as an argument. It's set to `true`, but had it been set to `false`, there would have been no output.

The examples presented so far expect that the code will be recompiled to turn debugging on and off. In a development environment, this is fine. However, in production, such luxury is not likely to be available. That's why the example in Listing 31.5 uses the `BooleanSwitch` and `Trace` classes.

LISTING 31.5 Implementing Debugging with a Boolean Switch:

```
BooleanSwitchDemo.cs
```

```
using System;
using System.Diagnostics;

/// <summary>
///     BooleanSwitch Demo.
```

LISTING 31.5 continued

```
/// </summary>
class BooleanSwitchDemo
{
    BooleanSwitch traceOutput = new
        BooleanSwitch("TraceOutput", "Boolean Switch Demo");

    void SetupDebugListener()
    {
        TextWriterTraceListener myListener =
            new TextWriterTraceListener(Console.Out);

        Trace.Listeners.Add(myListener);
    }

    void CheckState()
    {
        Trace.WriteLineIf(traceOutput.Enabled,
            "Debug: Entered CheckState()");
    }

    static void Main(string[] args)
    {
        BooleanSwitchDemo bsd = new BooleanSwitchDemo();

        bsd.SetupDebugListener();
        bsd.CheckState();
    }
}
```

And the output is:

```
Debug: Entered CheckState()
```

The `CheckState()` method of Listing 31.5 is similar to the same method in Listing 31.3, except that the `WriteLineIf()` method uses the `Enabled` property of a `BooleanSwitch` object as its first parameter. The `BooleanSwitch` class is instantiated with a first parameter as the display name and a second parameter as a description.

An entry must be added to the program's configuration file to turn on tracing. Listing 31.6 shows how to add the `BooleanSwitch` display name entry into the configuration file. The configuration file must have the same name as the executable with a `.config` extension.

LISTING 31.6 BooleanSwitch entry in Configuration File: `BooleanSwitchDemo.config`

```
<configuration>
    <system.diagnostics>
```

LISTING 31.6 continued

```
<switches>
    <add name="TraceOutput" value="1" />
</switches>
</system.diagnostics>
</configuration>
```

LISTING 31.7 Compilation Instructions for Listing 31.5

```
csc /d:TRACE BooleanSwitchDemo.cs
```

Runtime Tracing

Runtime tracing is the capability to perform debug tracing while a program is running. Sometimes it's necessary to have more control over what debugging information is displayed. Specific types of problems often indicate what information should be displayed in trace output. The `TraceSwitch` class is similar to the `BooleanSwitch` class in that it allows you to create a configuration file or set an environment variable. However, its real value comes in being able to specify a finer degree of granularity in determining what information is displayed. The example in Listing 31.8 demonstrates how to use the `TraceSwitch` class.

LISTING 31.8 TraceSwitch Class Demo: `TraceSwitchDemo.cs`

```
using System;
using System.Diagnostics;

/// <summary>
///     TraceSwitch Demo.
/// </summary>
class TraceSwitchDemo
{
    public static TraceSwitch traceOutput = new
        TraceSwitch("TraceOutput", "TraceSwitch Demo");

    void SetupDebugListener()
    {
        TextWriterTraceListener myListner =
            new TextWriterTraceListener(Console.Out);

        Trace.Listeners.Add(myListner);
    }
}
```

LISTING 31.8 continued

```
void CheckState()
{
    Trace.WriteLineIf(traceOutput.TraceInfo,
        "Trace: Entered CheckState()");
}

static void Main(string[] args)
{
    TraceSwitchDemo tsd = new TraceSwitchDemo();

    tsd.SetupDebugListener();
    tsd.CheckState();
}
}
```

And here's the output:

```
Trace: Entered CheckState()
```

The implementation of the `TraceSwitch` is similar to the `BooleanSwitch`, except that the first parameter to the `WriteLineIf()` method in the `CheckState()` method is the `TraceInfo` property of the `TraceSwitch` class. This parameter can be any of the possible values corresponding to a member of the `TraceLevel` enum, shown in Table 31.1.

TABLE 31.1 `TraceLevel` Enum

<code>TraceLevel</code> enum	Description
Verbose	Output everything
Info	Output info, error, and warning
Warning	Output error and warning
Error	Output error
Off	Output nothing

`TraceSwitch` must be set in a configuration file. Values may be from `0` to `4` with `Verbose` equal to `4` and descending to `Off`, which is equal to `0`. It's possible to create a custom switch by inheriting the `Switch` class and defining Boolean properties with your own unique names that map to the available members of the `TraceLevel` enum. The configuration file in Listing 31.9 has `TraceOutput` set to `3`, which causes evaluation of `TraceInfo` to return `true`.

LISTING 31.9 TraceSwitch Entry in Config File: TraceSwitchDemo.config

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="TraceOutput" value="3" />
    </switches>
  </system.diagnostics>
</configuration>
```

LISTING 31.10 Compilation Instructions for Listing 31.8

```
csc /d:TRACE TraceSwitchDemo.cs
```

Making Assertions

Another common debugging task is to check the state of a program at various intervals for logical consistency. This is performed with the `Debug.Assert()` method. By sprinkling `Assert()` methods at strategic points in a routine, such as preconditions, intermediate state, and post-conditions, you can verify that routine's logical consistency. Whenever the assertion proves false, a given message is displayed in the form of a message box. Listing 31.11 has a simple program demonstrating the mechanics of the `Assert()` method.

LISTING 31.11 Assertion Demonstration: AssertDemo.cs

```
using System;
using System.Diagnostics;

/// <summary>
///   Assertion Demonstration.
/// </summary>
class AssertDemo
{
    static void Main(string[] args)
    {
        decimal profit = -0.01m;

        // do some calculations

        Debug.Assert(profit >= 0.0m,
                    "Illogical Negative Profit Calculation");
    }
}
```

The example in Listing 31.11 simulates some fictitious profit calculation that should never return a negative result. The `Debug.Assert()` method takes two parameters. The first is the logical condition to check, which should evaluate to a Boolean `true` or `false`. In this case, it's making sure the profit is always zero or greater. The second parameter is the message to be displayed. The example forces the assertion to evaluate to `false`, displaying the message shown in Figure 31.1.

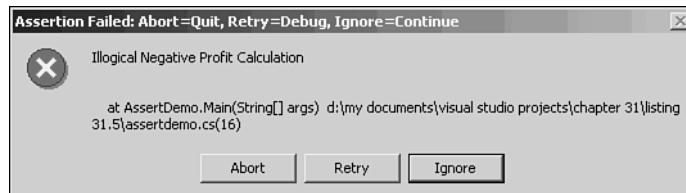
Assertions are designed to work only in debugging mode. Therefore, you will want to add a `/define` switch to the command-line when debugging. This program can be compiled with the command line in Listing 31.12.

LISTING 31.12 Compilation Instructions for Listing 31.11

```
csc /d:DEBUG AssertDemo.cs
```

FIGURE 31.1

Assertion message box.



Summary

Once appropriate statements and methods are in place, runtime debugging can make program verification more efficient by watching console printouts or viewing log files for pertinent results. Runtime debugging can be turned on and off with conditional attributes, specialized output methods that accept Boolean parameters, command line options, and preprocessing directives.

The `Debug` class is effective in development environments where the debugging code will be removed for deployment. Alternatively, the `Trace` class would be the best decision for situations where code should be deployed with a debugging capability.

Runtime debugging in trace-enabled code can be controlled with Boolean switches or multilevel trace switches. Each option provides a means of controlling the level of debugging with less disruption to a customer.

The `Debug.Assert()` method assists in verifying the logical consistency of an application during debugging. When a specified constraint fails, the `Assert()` method notifies the user with a message box displaying information about the reason for the failure.

Runtime detection of program errors is an important capability. Similarly, it's important to monitor the performance of a program. The next chapter, "Performance Monitoring," shows how to capture runtime performance of an application.

31

RUNTIME
DEBUGGING