LAB 1: Introduction to the Visual C# 2005 Express Edition IDE

OBJECTIVES

In this lab, you will learn:

- The basics of the Visual Studio Integrated Development Environment (IDE) that assists you in writing, running and debugging your Visual C# programs.
- Visual Studio's help features.
- Key commands contained in the IDE's menus and toolbars.
- The purpose of the various kinds of windows in the Visual Studio 2005 IDE.

1. Introduction

Visual Studio® 2005 is Microsoft's Integrated Development Environment (IDE) for creating, running and debugging programs (also called applications) written in a variety of .NET programming languages. In this chapter, we provide an overview of the Visual Studio 2005 IDE and demonstrate how to create a simple Visual C# program by dragging and dropping predefined building blocks into place, a technique called visual programming.

2. Overview of the Visual Studio 2005 IDE

To start Microsoft Visual C# 2005 Express Edition in Windows XP, select Start > All Programs > Visual C# 2005 Express Edition

To begin programming in Visual C#, you must create a new project or open an existing one. There are two ways to create a new project or open an existing project. You can select either File > New Project... or File > Open Project... from the File menu, which creates a new project or opens an existing project, respectively. From the Start Page, under the Recent Projects section, you can also click the links Create: Project or Open: Project/Solution. A project is a group of related files, such as the Visual C# code and any images that might make up a program. Visual Studio 2005 organizes programs into projects and solutions, which contain one or more projects. Multiple-project solutions are used to create large-scale programs. Each of the programs we create in this lab consists of a single project.

Select File > New Project... or the Create: Project... link on the Start Page to display the New Project dialog

Visual Studio provides templates for several project types. Templates are the project types users can create in Visual C# Windows applications, console applications and others. In this lab, we focus on Windows applications. . A Windows application is a

program that executes within a Windows operating system (e.g., Windows 2000 or Windows XP) and typically has a graphical user interface (GUI) the visual part of the program with which the user interacts. Windows applications include Microsoft software products like Microsoft Word, Internet Explorer and Visual Studio; software products created by other vendors; and customized software that you and other programmers create.

By default, Visual Studio assigns the name WindowsApplication1 to the new project and solution. Soon you will change the name of the project and the location where it is saved. Click OK to display the IDE in design view, which contains all the features necessary for you to begin creating programs. The design view portion of the IDE is also known as the Windows Form Designer.

The gray rectangle titled Form1 (called a Form) represents the main window of the Windows application that you are creating. C# applications can have multiple Forms (windows); however, most of the applications you will create in this lab use only one Form. Later in the lab, you will learn how to customize the Form by adding controls (i.e., reusable components) to a program in this case, a *Label* and a *PictureBox*. A Label typically contains descriptive text (e.g., "Welcome to Visual C#!"), and a PictureBox displays images. Visual Studio has over 65 preexisting controls you can use to build and customize your programs. In addition to the controls provided with Visual C# 2005 Express or Visual Studio 2005, there are many other controls available from third parties.

As you begin programming, you will work with controls that are not necessarily part of your program. As you place controls on the form, you can modify their properties by entering alternative text in a text box or selecting options then pressing a button, such as OK or Cancel.

Collectively, the Form and controls constitute the program's GUI. Users enter data (input) into the program in a variety of ways, including typing at the keyboard, clicking the mouse buttons and typing into GUI controls, such as TextBoxes. Programs use the GUI to display instructions and other information (output) for users to read. For example, the New Project dialog presents a GUI where the user clicks the mouse button to select a project template, and then inputs a project name from the keyboard.

The name of each open document is listed on a tab, the open documents are Form1.cs [Design] and the Start Page. To view a document, click its tab. Tabs facilitate easy access to multiple open documents. The active tab (the tab of the document currently displayed in the IDE) is displayed in bold text (e.g., **Form1.cs [Design]**) and is positioned in front of all the other tabs.

3. Menu Bar and Toolbar

Commands for managing the IDE and for developing, maintaining and executing programs are contained in menus, which are located on the menu bar of the IDE. Note that the set of menus displayed changes based on what you are currently doing in the IDE. Menus contain groups of related commands (also called menu items) that, when selected, cause the IDE to perform specific actions (e.g., open a window, save a file, print a file and execute a program). For example, new projects can be created by selecting File > New Project....

Menu	Description
File	Contains commands for opening, closing, adding and saving projects, as well as printing project data and exiting Visual Studio.
Edit	Contains commands for editing programs, such as cut, copy, paste, undo, redo, delete, find and select.
View	Contains commands for displaying windows (e.g., Solution Explorer, Toolbox, Properties window) and for adding toolbars to the IDE.
Project	Contains commands for managing projects and their files.
Build	Contains commands for compiling a program.
Debug	Contains commands for debugging (i.e., identifying and correcting problems in a program) and running a program.
Data	Contains commands for interacting with databases (i.e., organized collections of data stored on computers.
Format	Contains commands for arranging and modifying a form's controls. Note that the Format menu appears only when a GUI component is selected in Design view.
Tools	Contains commands for accessing additional IDE tools (e.g., the Toolbox) and options that enable you to customize the IDE.
Window	Contains commands for arranging and displaying windows.
Community	Contains commands for sending questions directly to Microsoft, checking question status, sending feedback on Visual C# and searching the CodeZone developer center and the Microsoft developers community site.
Help	Contains commands for accessing the IDE's help features.

Table 1-1: Summary of Visual Studio 2005 IDE menus.

Rather than navigating the menus from the menu bar, you can access many of the more common commands from the toolbar, which contains graphics, called icons, which graphically represent commands. [By default, the standard toolbar is displayed when you run Visual Studio for the first time; it contains icons for the most commonly used commands, such as opening a file, adding an item to a project, saving and running. Some commands are initially disabled (i.e., unavailable to use). These commands, which are initially grayed out, are enabled by Visual Studio only when they are necessary. For example, Visual Studio enables the command for saving a file once you begin editing the file.

You can customize the IDE by adding more toolbars. Select View > Toolbars. Each toolbar you select will be displayed with the other toolbars at the top of the Visual Studio window. Another way in which you can add toolbars to your IDE is through selecting Tools > Customize. Then, under the Toolbars tab, select the additional toolbars you would like to have appeared in the IDE.

To execute a command via the toolbar, click its icon. Some icons contain a down arrow that, when clicked, displays a related command or commands

It is difficult to remember what each of the icons on the toolbar represents. Positioning the mouse pointer over an icon highlights it and, after a brief delay, displays a description of the icon called a tool tip. Tool tips help novice programmers become familiar with the IDE's features and serve as useful reminders of each toolbar icon's functionality.

4. Navigating the Visual Studio 2005 IDE

The IDE provides windows for accessing project files and customizing controls. In this lab, we introduce several windows that you will use frequently when developing Visual C# programs. These windows can be accessed via toolbar icons or by selecting the name of the desired window in the View menu.

Visual Studio provides a space-saving feature called auto-hide. When auto-hide is enabled, a tab appears along either the left or right edge of the IDE window. This tab contains one or more icons, each of which identifies a hidden window. Placing the mouse pointer over one of these icons displays that window. The window is hidden again when the mouse pointer is moved outside of the window's area. To "pin down" a window (i.e., to disable auto-hide and keep the window open), click the pin icon. Note that when auto-hide is enabled, the pin icon is horizontal, whereas when a window is "pinned down," the pin icon is vertical.

4.1. Solution Explorer

The Solution Explorer window provides access to all of the files in a solution. If the Solution Explorer window is not shown in the IDE, select View > Solution Explorer or click the Solution Explorer icon. When you first open Visual Studio, the Solution

Explorer is empty; there are no files to display. Once you open a solution, the Solution Explorer displays the contents of the solution and its projects or when you create a new project, its contents are displayed.

The solution's startup project is the project that runs when the program executes. If you have multiple projects in a given solution, you can specify the startup project by right-clicking the project name in the Solution Explorer window, then selecting Set as StartUp Project. For a single-project solution, the startup project is the only project (in this case, WindowsApplication1) and the project name appears in bold text in the Solution Explorer window. All of the programs discussed in this lab are single-project solutions. For programmers using Visual Studio for the first time, the Solution Explorer window lists only the Properties, References, Form1.cs and Program.cs files. The Solution Explorer window includes a toolbar that contains several icons.

(Visual C# files use the .cs filename extension, which is short for "C Sharp.")

By default, the IDE displays only files that you may need to edit other files generated by the IDE are hidden. When clicked, the Show all files icon displays all the files in the solution, including those generated by the IDE. The plus and minus boxes that appear can be clicked to expand and collapse the project tree, respectively. Click the plus box to the left of Properties to display items grouped under the heading to the right of the plus box; click the minus boxes to the left of Properties and References to collapse the tree from its expanded state. Other Visual Studio windows also use this plus-box/minus-box convention.

4.2. Toolbox

The Toolbox contains icons representing controls used to customize forms. Using visual programming, you can "drag and drop" controls onto the form, which is faster and simpler than building them by writing GUI code. Just as you do not need to know how to build an engine to drive a car, you do not need to know how to build controls to use them. Reusing pre-existing controls saves time and money when you develop programs. The wide variety of controls contained in the Toolbox is a powerful feature of the .NET FCL. You will use the Toolbox when you create your first program later in the lab.

The Toolbox contains groups of related controls. Examples of these groups, All Windows Forms, Common Controls, Containers, Menus & Toolbars, Data, Components, Printing, Dialogs and General. Again, note the use of plus and minus boxes to expand or collapse a group of controls. The Toolbox contains over 65 prebuilt controls for use in Visual Studio, so you may need to scroll through the Toolbox to view additional controls.

4.3. Properties Window

To display the Properties window if it is not visible, you can select View > Properties Window, click the Properties window icon or you can press the F4 key. The Properties window displays the properties for the currently selected Form, control or file in design

view. Properties specify information about the form or control, such as its size, color and position. Each form or control has its own set of properties; a property's description is displayed at the bottom of the Properties window whenever that property is selected.

In the form's Properties window, the left column lists the Form's properties; the right column displays the current value of each property. Icons on the toolbar sort the properties either alphabetically by clicking the Alphabetical icon or categorically by clicking the Categorized icon. You can sort the properties alphabetically in ascending or descending order, clicking the Alphabetical icon repeatedly toggles between sorting the properties from AZ and ZA. Sorting by category groups the properties according to their use (i.e., Appearance, Behavior, and Design). Depending on the size of the Properties window, some of the properties may be hidden from view on the screen, in which case, users can scroll through the list of properties. We show how to set individual properties later in this lab.

The Properties window is crucial to visual programming; it allows you to modify a control's properties visually, without writing code. You can see which properties are available for modification and, where appropriate, can learn the range of acceptable values for a given property. The Properties window displays a brief description of the selected property, helping you understand its purpose. A property can be set quickly using this window usually, only one click is required, and no code needs to be written.

At the top of the Properties window is the component selection drop-down list, which allows you to select the form or control whose properties you wish to display in the Properties window. Using the component selection drop-down list is an alternative way to display properties without selecting the actual form or control in the GUI.

5. Using Help

Visual Studio provides extensive help features. The Help menu commands are summarized in table 1-2.

Command	Description
How Do I	Contains links to relevant topics, including how to upgrade programs and learn more about Web services, architecture and design, files and I/O, data, debugging and more.
Search	Finds help articles based on search keywords.
Index	Displays an alphabetized list of topics you can browse.
Contents	Displays a categorized table of contents in which help articles are organized by topic.

Table 1-2. Help menu commands.

Dynamic help is an excellent way to get information quickly about the IDE and its features. It provides a list of articles pertaining to the "current content" (i.e., the selected items). To open the Dynamic Help window, select Help > Dynamic Help. Then, when you click a word or component (such as a form or control), links to help articles appear in the Dynamic Help window. The window lists help topics, code samples and other relevant information. There is also a toolbar that provides access to the How Do I, Search, Index and Contents help features.

Visual Studio also provides context-sensitive help, which is similar to dynamic help, except that it immediately displays a relevant help article rather than presenting a list of articles. To use context-sensitive help, click an item, such as the form, and press the F1 key.

The Help options can be set in the Options dialog (accessed by selecting Tools > Options...). To display all the settings that you can modify (including the settings for the Help options), make sure that the Show all settings checkbox in the lower-left corner of the dialog is checked. To change whether the Help is displayed internally or externally, select Help on the left, and then locate the Show Help using: drop-down list on the right. Depending on your preference, selecting External Help Viewer displays a relevant help article in a separate window outside the IDE (some programmers like to view Web pages separately from the project on which they are working in the IDE). Selecting Integrated Help Viewer displays a help article as a tabbed window inside the IDE.

LAB 2: Introduction to C# applications

Part 1:

1. Objectives

In this lab you will learn:

- To write simple C# applications using code rather than visual programming.
- To write statements that input and output data to the screen.
- To declare and use data of various types.
- To store and retrieve data from memory.
- To use arithmetic operators.
- To determine the order in which operators are applied.
- To write decision-making statements.
- To use relational and equality operators.
- To use message dialogs to display messages.

2. Introduction

We now introduce C# application programming, which facilitates a disciplined approach to application design. Most of the C# applications you will study in this lab process information and display results. In this lab, we introduce console applications those input and output text in a console window. In Microsoft Windows 95/98/ME, the console window is the MS-DOS prompt. In other versions of Microsoft Windows, the console window is the Command Prompt.

We begin with several examples that simply display messages on the screen. We then demonstrate an application that obtains two numbers from a user, calculates their sum and displays the result. You will learn how to perform

various arithmetic calculations and save the results for later use. Many applications contain logic that requires the application to make decisions the last example in this lab demonstrates decision-making fundamentals by showing you how to compare numbers and display messages based on the comparison results. For example, the application displays a message indicating that two numbers are equal only if they have the same value. We carefully analyze each example one line at a time.

3. A Simple C# Application: Displaying a Line of Text

Let us consider a simple application that displays a line of text. The application illustrates several important C# language features. C# uses notations that may look strange to nonprogrammers. For your convenience, each program we present in this lab includes line numbers, which are not part of actual C# code. We will soon see that line 10 does the real work of the application namely, displaying the phrase Welcome to C# Programming! on the screen. We now consider each line of the application; this is called a code walkthrough.

```
 1 // ex. 1.1: Welcome1.cs
 2 // Text-printing application.
 3 using System;
 4
 5 public class Welcome1
 6 {
 7    // Main method begins execution of C# application
 8    public static void Main( string[] args )
 9    {
10       Console.WriteLine( "Welcome to C# Programming!" );
11    } // end method Main
12 } // end class Welcome1
```

Line 1

begins with //, indicating that the remainder of the line is a comment.

A comment that begins with // is called a single-line comment, because it terminates at the end of the line on which it appears. A // comment also can

begin in the middle of a line and continue until the end of that line (as in lines 7, 11 and 12).

Delimited comments such as

```
/* This is a delimited
   comment. It can be
   split over many lines */
```

can be spread over several lines. This type of comment begins with the delimiter /* and ends with the delimiter */. All text between the delimiters is ignored by the compiler.

Line 2

```
// Text-printing application.
```

is a single-line comment that describes the purpose of the application.

Line 3

```
using System;
```

is a using directive that helps the compiler locate a class that is used in this application. A great strength of C# is its rich set of predefined classes that you can reuse. These classes are organized under namespaces named collections of related classes. Collectively, .NET's namespaces are referred to as the .NET Framework Class Library (FCL). Each using directive identifies predefined classes that a C# application should be able to use. The using directive in line 3 indicates that this example uses classes from the System namespace, which contains the predefined *Console class* (discussed shortly) used in Line 10, and many other useful classes.

This can be found in the Visual C# Express documentation under the Help menu. You can also place the cursor on the name of any .NET class or method, and then press the F1 key to get more information.

Line 4

is simply a blank line. Programmers use blank lines and space characters to make applications easier to read. Together, blank lines, space characters and tab characters are known as whitespace. (Space characters and tabs are known specifically as whitespace characters.) Whitespace is ignored by the compiler. In this and the next several labs, we discuss conventions for using whitespace to enhance application readability.

Line 5

```
public class Welcome1
```

begins a class declaration for the class Welcome1. Every application consists of at least one class declaration that is defined by you the programmer. These are known as user-defined classes. The class keyword introduces a class declaration and is immediately followed by the class name (Welcome1). Keywords (sometimes called reserved words) are reserved for use by C# and are always spelled with all lowercase letters. The complete list of C# keywords is shown in Table 1.1.

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

C# Keywords				
override	params	private	protected	public
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
volatile	while			

Table 1.1 C# keywords.

By convention, all class names begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`). This is frequently referred to as Pascal casing. A class name is an identifier series of characters consisting of letters, digits and underscores (_) that does not begin with a digit and does not contain spaces. Some valid identifiers are `Welcome1`, `identifier`, `_value` and `m_inputField1`. The name `7button` is not a valid identifier because it begins with a digit, and the name `input field` is not a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not the name of a class. C# is case sensitive that is, uppercase and lowercase letters are distinct, so `a1` and `A1` are different (but both valid) identifiers. Identifiers may also be preceded by the `@` character. This indicates that a word should be interpreted as an identifier, even if it is a keyword (e.g. `@int`). This allows C# code to use code written in other .NET languages where an identifier might have the same name as a C# keyword.

When you save your public class declaration in a file, the file name is usually the class name followed by the *.cs filename extension*. For our application, the file name is Welcome1.cs.

A left brace (in line 6), {, begins the body of every class declaration. A corresponding right brace (in line 12), }, must end each class declaration. Note that lines 7-11 are indented. This indentation is one of the spacing conventions mentioned earlier. We define each spacing convention as a Good Programming Practice.

Line 7

```
// Main method begins execution of C# application
```

is a comment indicating the purpose of lines 8 of the application.

Line 8

```
public static void Main( string[] args )
```

is the starting point of every application. The parentheses after the identifier *`Main`* indicate that it is an application building block called a method. Class declarations normally contain one or more methods. Method names usually follow the same Pascal casing capitalization conventions used for class names. For each application, exactly one of the methods in a class must be called `Main` (which is typically defined as shown in line 8); otherwise, the application will not execute. Methods are able to perform tasks and return information when they complete their tasks. Keyword `void` (line 8) indicates that this method will not return any information after it completes its task. Later, we will see that many methods do return information.

The left brace in line 9 begins the body of the method declaration. A corresponding right brace must end the method's body (line 11). Note that line 10 in the body of the method is indented between the braces.

Line 10

```
Console.WriteLine( "Welcome to C# Programming!" );
```

instructs the computer to perform an action namely, to print (i.e., display on the screen) the string of characters contained between the double quotation

marks. A string is sometimes called a character string, a message or a string literal. We refer to characters between double quotation marks simply as strings. Whitespace characters in strings are not ignored by the compiler.

Class `Console` provides standard input/output capabilities that enable applications to read and display text in the console window from which the application executes. The `Console.WriteLine` method displays (or prints) a line of text in the console window. The string in the parentheses in line 10 is the argument to the method. Method `Console.WriteLine` performs its task by displaying (also called outputting) its argument in the console window. When `Console.WriteLine` completes its task, it positions the screen cursor (the blinking symbol indicating where the next character will be displayed) at the beginning of the next line in the console window. (This movement of the cursor is similar to what happens when a user presses the Enter key while typing in a text editor the cursor moves to the beginning of the next line in the file.)

The entire line 10, including `Console.WriteLine`, the parentheses, the argument `"Welcome to C# Programming!"` in the parentheses and the semicolon (`;`), is called a statement. Each statement ends with a semicolon. When the statement in line 10 executes, it displays the message `Welcome to C# Programming!` in the console window. A method is typically composed of one or more statements that perform the method's task.

Some programmers find it difficult when reading or writing an application to match the left and right braces (`{` and `}`) that delimit the body of a class declaration or a method declaration. For this reason, some programmers include a comment after each closing right brace (`}`) that ends a method declaration and after each closing right brace that ends a class declaration. For example, line 11

```
} // end method Main
```

specifies the closing right brace of method `Main`, and line 12

```
} // end class Welcome1
```

specifies the closing right brace of class `Welcome1`. Each of these comments indicates the method or class that the right brace terminates. Visual Studio can help you locate matching braces in your code. Simply place the cursor next to one brace and Visual Studio will highlight the other.

4. Creating Simple Application in Visual C# Express

Creating the Console Application

After opening Visual C# 2005 Express, select File > New Project... to display the New Project dialog, then select the Console Application template. In the dialog's Name field, type `Welcome1`. Click OK to create the project. The IDE now contains the open console application, as shown in. Note that the editor window already contains some code provided by the IDE. The IDE inserts this extra code to help organize the application and to provide access to some common classes in the .NET Framework Class Libraryat this point in the book, this code is neither required nor relevant to the discussion of this application; delete all of it.

The code coloring scheme used by the IDE is called syntax-color highlighting and helps you visually differentiate application elements. Keywords appear in blue, and other text is black. When present, comments are green. In this book, we syntax shade our code similarly bold italic for keywords, italic for comments, bold gray for literals and constants, and black for other text. One example of a literal is the string passed to `Console.WriteLine` in line 10. You can customize the colors shown in the code editor by selecting Tools > Options.... This displays the Options dialog. Then click the plus sign, +, next to Environment and select Fonts and Colors. Here you can change the colors for various code elements.

Modifying the Editor Settings to Display Line Numbers

Visual C# Express provides many ways to personalize your coding experience. In this step, you will change the settings so that your code

matches that of this book. To have the IDE display line numbers, select Tools > Options.... In the dialog that appears, click the Show all settings checkbox on the lower left of the dialog, then click the plus sign next to Text Editor in the left pane and select All Languages. On the right, check the Line Numbers check box. Keep the Options dialog open.

Setting Code Indentation to Three Spaces per Indent

In the Options dialog that you opened in the previous step, click on the plus sign next to C# in the left pane and select Tabs. Enter 3 for both the Tab Size and Indent Size fields. Any new code you add will now use three spaces for each level of indentation. Click OK to save your settings, close the dialog and return to the editor window.

Changing the Name of the Application File

For applications we create in this lab, we change the default name of the application file (i.e., `Program.cs`) to a more descriptive name. To rename the file, click `Program.cs` in the Solution Explorer window. This displays the application file's properties in the Properties window. Change the File Name property to `Welcome1.cs`.

Writing Code

In the editor window, type the code. After you type (in line 10) the class name and a dot (i.e., `Console.`), a window containing a scrollbar is displayed . This IDE feature, called IntelliSense, lists a class's members, which include method names. As you type characters, Visual C# Express highlights the first member that matches all the characters typed, then displays a tool tip containing a description of that member. You can either type the complete member name (e.g., `WriteLine`), double click the member name in the member list or press the Tab key to complete the name. Once the complete name is provided, the *IntelliSense* window closes.

When you type the open parenthesis character, `(`, after `Console.WriteLine`, the Parameter Info window is displayed. This

window contains information about the method's parameters. As you will learn later, there can be several versions of a method that is, a class can define several methods that have the same name as long as they have different numbers and/or types of parameters. These methods normally all perform similar tasks. The Parameter Info window indicates how many versions of the selected method are available and provides up and down arrows for scrolling through the different versions. For example, there are 19 versions of the `WriteLine` method we use one of these 19 versions in our application. The Parameter Info window is one of the many features provided by the IDE to facilitate application development. In the next several chapters, you will learn more about the information displayed in these windows. The Parameter Info window is especially helpful when you want to see the different ways in which a method can be used. From the code , we already know that we intend to display one string with `WriteLine`, so because you know exactly which version of `WriteLine` you want to use, you can simply close the Parameter Info window by pressing the Esc key.

Saving the Application

Select File > Save All to display the Save Project dialog. In the Location text box, specify the directory where you want to save this project. We choose to save the project in the `MyProjects` directory on the `C:` drive. Select the Create directory for solution checkbox (to enable Visual Studio to create the directory if it does not already exist), and click Save.

Compiling and Running the Application

You are now ready to compile and execute your application. Depending on the type of application, the compiler may compile the code into files with a `.exe` (executable) extension, a `.dll` (dynamic link library) extension or one of several other extensions. Such files are called assemblies and are the packaging units for compiled C# code. These assemblies contain the Microsoft Intermediate Language (MSIL) code for the application.

To compile the application, select Build > Build Solution. If the application contains no syntax errors, your console application will compile into an executable file (named `Welcome1.exe`, in the project's directory). To execute this console application (i.e., `Welcome1.exe`), select Debug > Start Without Debugging (or type <Ctrl> F5), which invokes the `Main` method. The statement in line 10 of `Main` displays `Welcome to C# Programming!`. Note that the results are displayed in a console window. Leave the application open in Visual C# Express; we will go back to it later in this section.

Running the Application from the Command Prompt

As we mentioned at the beginning of the chapter, you can execute applications outside the IDE in a Command Prompt. This is useful when you simply want to run an application rather than open it for modification. To open the Command Prompt, select Start > All Programs > Accessories > Command Prompt. [Note: Windows 2000 users should replace All Programs with Programs.] The window displays copyright information, followed by a prompt that indicates the current directory. By default, the prompt specifies the current user's directory on the local machine. On your machine, the folder name will be replaced with your username. Enter the command `cd` (which stands for "change directory"), followed by the `/d` flag (to change drives if necessary), then the directory where the application's `.exe` file is located (i.e., the `Release` directory of your application). For example, the command `cd /d C:\MyProjects\Welcome1\Welcome1\bin\Release` changes the current directory, to the `Welcome1` application's `Release` directory on the `C:` drive. The next prompt displays the new directory. After changing to the proper directory, you can run the compiled application by entering the name of the `.exe` file (i.e., `Welcome1`). The application will run to completion, then the prompt will display again, awaiting the next command. To close the Command Prompt, type `exit` and press Enter.

Note that Visual C# 2005 Express maintains a `Debug` and a `Release` directory in each project's `bin` directory. The `Debug` directory contains a version of the application that can be used with the debugger. The `Release` directory contains an optimized version that you could provide to your clients. In the complete Visual Studio 2005, you can select the specific version you wish to build from the Solution Configurations drop-down list in the toolbars at the top of the IDE. The default is the `Debug` version.

Syntax Errors, Error Messages and the Error List Window

Go back to the application in Visual C# Express. When you type a line of code and press the Enter key, the IDE responds either by applying syntax-color highlighting or by generating a syntax error, which indicates a violation of Visual C#'s rules for creating correct applications (i.e., one or more statements are not written correctly). Syntax errors occur for various reasons, such as missing parentheses and misspelled keywords.

When a syntax error occurs, the IDE underlines the error in red and provides a description of the error in the Error List window. If the Error List window is not visible in the IDE, select View > Error List to display it. we intentionally omitted the first parenthesis in line 10. The first error contains the text "; expected" and specifies that the error is in column 25 of line 10. This error message appears when the compiler thinks that the line contains a complete statement, followed by a semicolon, and the beginning of another statement. The second error contains the same text, but specifies that this error is in column 54 of line 10 because the compiler thinks that this is the end of the second statement. The third error has the text "Invalid expression term ')'" because the compiler is confused by the unmatched right parenthesis. Although we are attempting to include only one statement in line 10, the missing left parenthesis causes the compiler to incorrectly assume that there is more than one statement on that line, to misinterpret the right parenthesis and to generate three error messages.

LAB 3: Introduction to C# applications

Part 2:

5. Another C# Application: Adding Integers

Our next application reads (or inputs) two integers (whole numbers, like 22, 7, 0 and 1024) typed by a user at the keyboard, computes the sum of the values and displays the result. This application must keep track of the numbers supplied by the user for the calculation later in the application. Applications remember numbers and other data in the computer's memory and access that data through application elements called variables. The application demonstrates these concepts. In the sample output, we highlight data the user enters at the keyboard in bold.

```
 1 // Fig. 1.2: Addition.cs
 2 // Displaying the sum of two numbers input from the
keyboard.
 3 using System;
 4
 5 public class Addition
 6 {
 7    // Main method begins execution of C# application
 8    public static void Main( string[] args )
 9    {
10       int number1; // declare first number to add
11       int number2; // declare second number to add
12       int sum; // declare sum of number1 and number2
13
14       Console.Write( "Enter first integer: " ); //
prompt user
15       // read first number from user
16       number1 = Convert.ToInt32( Console.ReadLine() );
17
18       Console.Write( "Enter second integer: " ); //
prompt user
19       // read second number from user
20       number2 = Convert.ToInt32( Console.ReadLine() );
21
```

```
22        sum = number1 + number2; // add numbers
23
24        Console.WriteLine( "Sum is {0}", sum ); // display
sum
25     } // end method Main
26 } // end class Addition
```

Lines 12

```
// Fig. 1.2: Addition.cs
// Displaying the sum of two numbers input from the
keyboard.
```

state the figure number, file name and purpose of the application.

Line 5

```
public class Addition
```

begins the declaration of class `Addition`. Remember that the body of each class declaration starts with an opening left brace (line 6), and ends with a closing right brace (line 26).

The application begins execution with method `Main` (lines 8-25). The left brace (line 9) marks the beginning of `Main`'s body, and the corresponding right brace (line 25) marks the end of `Main`'s body. Note that method `Main` is indented one level within the body of class `Addition` and that the code in the body of `Main` is indented another level for readability.

Line 10

```
int number1; // declare first number to add
```

is a variable declaration statement (also called a declaration) that specifies the name and type of a variable (`number1`) that is used in this application.

A variable is a location in the computer's memory where a value can be stored for use later in an application. All variables must be declared with a name and a type before they can be used. A variable's name enables the application to access the value of the variable in memory the name can be any valid identifier. A variable's type specifies what kind of information is stored at that location in memory. Like other statements, declaration statements end with a semicolon (`;`).

The declaration in line 10 specifies that the variable named `number1` is of type `int` it will hold integer values (whole numbers such as `7`, `11`, `0` and `31914`). The range of values for an `int` is 2,147,483,648 (`int.MinValue`) to +2,147,483,647 (`int.MaxValue`). We will soon discuss types `float`, `double` and `decimal`, for specifying real numbers, and type `char`, for specifying characters. Real numbers contain decimal points, as in `3.4`, `0.0` and `11.19`. Variables of type `float` and `double` store approximations of real numbers in memory. Variables of type `decimal` store real numbers precisely (to 2829 significant digits), so `decimal` variables are often used with monetary calculations. Variables of type `char` represent individual characters, such as an uppercase letter (e.g., `A`), a digit (e.g., `7`), a special character (e.g., `*` or `%`) or an escape sequence (e.g., the newline character, `\n`). Types such as `int`, `float`, `double`, `decimal` and `char` are often called simple types. Simple-type names are keywords and must appear in all lowercase letters.

The variable declaration statements at lines 11-12

```
int number2; // declare second number to add
int sum; // declare sum of number1 and number2
```

similarly declare variables `number2` and `sum` to be of type `int`.

Variable declaration statements can be split over several lines, with the variable names separated by commas (i.e., a comma-separated list of variable names). Several variables of the same type may be declared in one

declaration or in multiple declarations. For example, lines 1012 can also be written as follows:

```
int number1, // declare first number to add
    number2, // declare second number to add
    sum; // declare sum of number1 and number2
```

Line 14

```
Console.Write( "Enter first integer: " ); // prompt
user
```

uses *`Console.Write`* to display the message `"Enter first integer: "`. This message is called a prompt because it directs the user to take a specific action.

Line 16

```
number1 = Convert.ToInt32( Console.ReadLine() );
```

works in two steps. First, it calls the `Console`'s `ReadLine` method. This method waits for the user to type a string of characters at the keyboard and press the Enter key to submit the string to the application. Then, the string is used as an argument to the `Convert` class's `ToInt32` method, which converts this sequence of characters into data of an type `int`. As we mentioned earlier in this chapter, some methods perform a task then return the result of that task. In this case, method `ToInt32` returns the int representation of the user's input.

Technically, the user can type anything as the input value. `ReadLine` will accept it and pass it off to the `ToInt32` method. This method assumes that the string contains a valid integer value. In this application, if the user types a noninteger value, a runtime logic error will occur and the application will terminate.

In line 16, the result of the call to method `ToInt32` (an `int` value) is placed in variable `number1` by using the assignment operator, =. The statement is read as "`number1` gets the value returned by `Convert.ToInt32`." Operator = is called a binary operator because it has two operands`number1` and the result of the method call `Convert.ToInt32`. This statement is called an assignment statement because it assigns a value to a variable. Everything to the right of the assignment operator, =, is always evaluated before the assignment is performed.

Line 18

```
Console.Write( "Enter second integer: " ); //
prompt user
```

prompts the user to enter the second integer. Line 20

```
number2 = Convert.ToInt32( Console.ReadLine() );
```

reads a second integer and assigns it to the variable `number2`.

Line 22

```
sum = number1 + number2; // add numbers
```

is an assignment statement that calculates the sum of the variables `number1` and `number2` and assigns the result to variable `sum` by using the assignment operator, =. The statement is read as "`sum` gets the value of `number1 + number2`." Most calculations are performed in assignment statements. When the application encounters the addition operator, it uses the values stored in the variables `number1` and `number2` to perform the calculation. In the preceding statement, the addition operator is a binary operator its two operands are `number1` and `number2`. Portions of statements that contain calculations are called expressions. In fact, an expression is any portion of a statement that has a value associated with it. For example, the value of the expression `number1 + number2` is the

sum of the numbers. Similarly, the value of the expression `Console.ReadLine()` is the string of characters typed by the user.

After the calculation has been performed, line 24

```
Console.WriteLine( "Sum is {0}", sum ); // display sum
```

uses method `Console.WriteLine` to display the `sum`. The format item `{0}` is a placeholder for the first argument after the format string. Other than the `{0}` format item, the remaining characters in the format string are all fixed text. So method `WriteLine` displays `"Sum is "`, followed by the value of `sum` (in the position of the `{0}` format item) and a newline.

Calculations can also be performed inside output statements. We could have combined the statements in lines 22 and 24 into the statement

```
Console.WriteLine( "Sum is {0}", ( number1 + number2 ) );
```

The parentheses around the expression `number1 + number2` are not required they are included to emphasize that the value of the expression `number1 + number2` is output in the position of the `{0}` format item.

6. Arithmetic

Most applications perform arithmetic calculations. The arithmetic operators are summarized in Fig. 1.3. Note the use of various special symbols not used in algebra. The asterisk (*) indicates multiplication, and the percent sign (%) is the remainder operator (called modulus in some languages), which we will discuss shortly. The arithmetic operators in Fig. 1.3 are binary operators for example, the expression f + 7 contains the binary operator + and the two operands f and 7.

Figure 1.3. Arithmetic operators.

C# operation	Arithmetic operator	Algebraic expression	C# expression
Addition	+	f + 7	f + 7
Subtraction		p c	p - c
Multiplication	*	b · m	b * m
Division	/	x / y or or x ÷ y	x / y
Remainder	%	r mod s	r % s

Integer division yields an integer quotient for example, the expression 7 / 4 evaluates to 1, and the expression 17 / 5 evaluates to 3. Any fractional part in integer division is simply discarded (i.e., truncated)no rounding occurs. C# provides the remainder operator, %, which yields the remainder after division. The expression x % y yields the remainder after x is divided by y. Thus, 7 % 4 yields 3, and 17 % 5 yields 2. This operator is most commonly used with integer operands, but can also be used with floats, doubles, and decimals. In the chapter's exercises and in later chapters, we consider several interesting applications of the remainder operator, such as determining whether one number is a multiple of another.

Arithmetic expressions must be written in straight-line form to facilitate entering applications into the computer. Thus, expressions such as "a divided by b" must be written as a / b, so that all constants, variables and operators appear in a straight line.

Parentheses are used to group terms in C# expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity b + c, we write

a * (b + c)

If an expression contains nested parentheses, such as

((a + b) * c)

the expression in the innermost set of parentheses (a + b in this case) is evaluated first.

C# applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those followed in algebra (1.4):

1. Multiplication, division and remainder operations are applied first. If an expression contains several such operations, the operators are applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

Figure 1.4. Precedence of arithmetic operators.

Operator(s)	Operation(s)	Order of evaluation (associativity)
Evaluated first		
* / %	Multiplication Division Remainder	If there are several operators of this type, they are evaluated from left to right.
Evaluated next		
+ -	Addition Subtraction	If there are several operators of this type, they are evaluated from left to right.

These rules enable C# to apply operators in the correct order. When we say that operators are applied from left to right, we are referring to their associativity. You will see that some operators associate from right to left. Figure 1.4 summarizes these rules of operator precedence. The table will be expanded as additional operators are introduced.

Now let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C# equivalent. The following is an example of an arithmetic mean (average) of five terms:

m = (a + b + c + d + e) / 5;

The parentheses are required because division has higher precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided

by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates as

The following is an example of the equation of a straight line:

Algebra: `y = mx + b;`

```
C#:     y = m * x + b;
```

No parentheses are required. The multiplication operator is applied first because multiplication has a higher precedence than addition. The assignment occurs last because it has a lower precedence than multiplication or addition.

7. Decision Making: Equality and Relational Operators

A condition is an expression that can be either true or false. This section introduces a simple version of C#'s if statement that allows an application to make a decision based on the value of a condition. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an *if statement* is true, the body of the if statement executes. If the condition is false, the body does not execute. We will see an example shortly.

Conditions in if statements can be formed by using the equality operators (==, and !=) and relational operators (>, <, >= and <=) summarized in Fig. 1.5. The two equality operators (== and !=) each have the same level of precedence, the relational operators (>, <, >= and <=) each have the same level of precedence, and the equality operators have lower precedence than the relational operators. They all associate from left to right.

Figure 1.5. Equality and relational operators.

(This item is displayed on page 103 in the print version)

Standard algebraic equality and relational operators	**C# equality or relational operator**	**Sample C# condition**	**Meaning of C# condition**
Equality operators			
=	= =	x = = y	x is equal to y
	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
	>=	x >= y	x is greater than or equal to y
	<=	x <= y	x is less than or equal to y

The application of Fig. 1.6 uses six if statements to compare two integers entered by the user. If the condition in any of these if statements is true, the assignment statement associated with that if statement executes. The application uses the Console class to prompt for and read two lines of text

from the user, extracts the integers from that text with the ToInt32 method of class Convert, and stores them in variables number1 and number2. Then the application compares the numbers and displays the results of the comparisons that are true.

Figure 1.6. Comparing integers using if statements, equality operators and relational operators.

```
1 // Fig. 1.6: Comparison.cs
2 // Comparing integers using if statements, equality operators,
3 // and relational operators.
4 using System;
5
6 public class Comparison
7 {
8    // Main method begins execution of C# application
9    public static void Main( string[] args )
10   {
11      int number1; // declare first number to compare
12      int number2; // declare second number to compare
13
14      //prompt user and read first number
15      Console.Write( "Enter first integer: " );
16      number1 = Convert.ToInt32( Console.ReadLine() );
17
```

```
18      //prompt user and read second number
19      Console.Write( "Enter second integer: " );
20      number2 = Convert.ToInt32( Console.ReadLine() );
21
22      if ( number1 == number2 )
23         Console.WriteLine( "{0} == {1}", number1, number2 );
24
25      if ( number1 != number2 )
26         Console.WriteLine( "{0} != {1}", number1, number2 );
27
28      if ( number1 < number2 )
29         Console.WriteLine( "{0} < {1}", number1, number2 );
30
31      if ( number1 > number2 )
32         Console.WriteLine( "{0} > {1}", number1, number2 );
33
34      if ( number1 <= number2 )
35         Console.WriteLine( "{0} <= {1}", number1, number2 );
36
37      if ( number1 >= number2 )
38         Console.WriteLine( "{0} >= {1}", number1, number2 );
39   } // end method Main
```

```
40 } // end class Comparison
```

The declaration of class Comparison begins at line 6

public class Comparison

The class's Main method (lines 939) begins the execution of the application.

Lines 11-12

int number1; // declare first number to compare

int number2; // declare second number to compare

declare the int variables used to store the values entered by the user.

Lines 14-16

// prompt user and read first number

Console.Write("Enter first integer: ");

number1 = Convert.ToInt32(Console.ReadLine());

prompt the user to enter the first integer and input the value. The input value is stored in variable number1.

Lines 18-20

// prompt user and read second number

Console.Write("Enter second integer: ");

number2 = Convert.ToInt32(Console.ReadLine());

perform the same task, except that the input value is stored in variable number2.

Lines 22-23

if (number1 == number2)

  Console.WriteLine("{0} == {1}", number1, number2);

compare the values of the variables number1 and number2 to determine whether they are equal. An if statement always begins with keyword if, followed by a condition in parentheses. An if statement expects one statement in its body. The indentation of the body statement shown here is not required, but it improves the code's readability by emphasizing that the statement in line 23 is part of the if statement that begins in line 22. Line 23 executes only if the numbers stored in variables number1 and number2 are equal (i.e., the condition is true). The if statements in lines 2526, 2829, 3132, 3435 and 3738 compare number1 and number2 with the operators !=, <, >, <= and >=, respectively. If the condition in any of the if statements is true, the corresponding body statement executes.

Note that there is no semicolon (`;`) at the end of the first line of each `if` statement. Such a semicolon would result in a logic error at execution time. For example,

```
if ( number1 == number2 ); // logic error
   Console.WriteLine( "{0} == {1}", number1,
number2 );
```

would actually be interpreted by C# as

```
if ( number1 == number2 )
   ; // empty statement

Console.WriteLine( "{0} == {1}", number1, number2
);
```

where the semicolon in the line by itself called the empty statement is the statement to execute if the condition in the `if` statement is true. When the empty statement executes, no task is performed in the application. The application then continues with the output statement, which always executes, regardless of whether the condition is true or false, because the output statement is not part of the `if` statement.

Note the use of whitespace in Fig. 1.6. Recall that whitespace characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So statements may be split over several lines and may be spaced according to your preferences without affecting the meaning of an application. It is incorrect to split identifiers, strings, and multicharacter operators (like >=). Ideally, statements should be kept small, but this is not always possible.

Figure 1.7 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. All these operators, with the exception of the assignment operator, =, associate from left to right. Addition is left associative, so an expression like `x + y + z` is evaluated as if it had been written as `( x + y ) + z`. The assignment operator, =, associates from right to left, so an expression like `x = y = 0` is evaluated as if it had been written as `x = ( y = 0 )`, which, as you will soon see, first assigns the value `0` to variable `y` and then assigns the result of that assignment, `0`, to `x`.

Figure 1.7. Precedence and associativity of operations discussed.

Operators				**Associativity**	**Type**
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Exercises

1. Write an application that asks the user to enter two integers, obtains them from the user and displays the larger number followed by the words "is larger". If the numbers are equal, print the message "These numbers are equal."
2. Write an application that reads five integers, then determines and prints the largest and smallest integers in the group.
3. Write an application that inputs five numbers and determines and prints the number of negative numbers input, the number of positive numbers input and the number of zeros input.

Lab 4: Introducing Classes and Objects

Part 1:

The class is the foundation of C# because it defines the nature of an object. Furthermore, the class forms the basis for object-oriented programming. Within a class are defined both code and data. Because classes and objects are fundamental to C#, they constitute a large topic.

Class Fundamentals

A class is a template that defines the form of an object. It specifies both the data and the code that will operate on that data. C# uses a class specification to construct *objects.* Objects are *instances* of a class. Thus, a class is essentially a set of plans that specify how to build an object. It is important to be clear on one issue: A class is a logical abstraction. It is not until an object of that class has been created that a physical representation of that class exists in memory.

The General Form of a Class

When you define a class, you declare the data that it contains and the code that operates on it. While very simple classes might contain only code or only data, most real-world classes contain both.

In general terms, data is contained in *data members* defined by the class, and code is contained in *function members*. It is important to state at the outset that C# defines several specific flavors of data and function members. For example, data members (also called *fields*) include instance variables and static variables. Function members include methods, constructors, destructors, indexers, events, operators, and properties. For now, we will limit our discussion of the class to its essential elements: instance variables and methods. Later in this lab, constructors and destructors are discussed. The other types of members are described in later labs.

A class is created by use of the keyword **class**. Here is the general form of a simple **class** definition that contains only instance variables and methods:

```
class classname{
  // declare instance variables
  access type var1;
  access type var2;
  // ...
  access type varN;

  // declare methods
  access ret-type method1(parameters) {
     // body of method
  }
  access ret-type method2(parameters) {
     // body of method
  }
     // ...
  access ret-type methodN(parameters) {
     // body of method
  }
}
```

Notice that each variable and method declaration is preceded with *access*. Here, *access* is an access specifier, such as **public**, which specifies how the member can be accessed. Class members can be private to a class or more accessible. The access specifier determines what type of access is allowed. The access specifier is optional and if absent, then the member is private to the class. Members with private access can be used only by other members of their class. For the examples in this lab, all members will be specified as **public**, which means that they can be used by all other code—even code defined outside the class.

Note *In addition to an access specifier, the declaration of a class member can also contain one or more type modifiers.*

Although there is no syntactic rule that enforces it, a well-designed class should define one and only one logical entity. For example, a class that stores names and telephone numbers will not normally also store information about the stock market, average rainfall, sunspot cycles, or other unrelated information. The point here is that a well-designed class groups logically connected information. Putting unrelated information into the same class will quickly destructure your code.

Up to this point, the classes that we have been using have only had one method: **Main( )**. However, notice that the general form of a class does not specify a **Main( )** method. A **Main( )** method is required only if that class is the starting point for your program.

Defining a Class

To illustrate classes, we will be evolving a class that encapsulates information about buildings, such as houses, stores, offices, and so on. This class is called **Building**, and it will store three items of information about a building: the number of floors, the total area, and the number of occupants.

The first version of **Building** is shown here. It defines three instance variables: **floors**, **area**, and **occupants**. Notice that **Building** does not contain any methods. Thus, it is currently a data-only class. (Subsequent sections will add methods to it.)

```
class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants
}
```

The instance variables defined by **Building** illustrate the way that instance variables are declared in general. The general form for declaring an instance variable is shown here:

access type var-name;

Here, *access* specifies the access, *type* specifies the type of variable, and *var-name* is the variable's name. Thus, aside from the access specifier, you declare an instance variable in the same way that you declare local variables. For **Building**, the variables are preceded by the **public** access modifier. As explained, this allows them to be accessed by code outside of **Building**.

A **class** definition creates a new data type. In this case, the new data type is called **Building**. You will use this name to declare objects of type **Building**. Remember that a **class** declaration is only a type description; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Building** to come into existence.

To actually create a **Building** object, you will use a statement like the following:

Building house = new Building(); // create an object of type building

After this statement executes, **house** will be an instance of **Building**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Building** object will contain its own copies of the instance variables **floors**, **area**, and **occupants**. To access these variables, you will use the *dot* (**.**) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Thus, the object is specified on the left, and the member is put on the right. For example, to assign the **floors** variable of **house** the value 2, use the following statement:

house.floors = 2;

In general, you can use the dot operator to access both instance variables and methods.

Here is a complete program that uses the **Building** class:

```
// A program that uses the Building class.

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants
}

// This class declares an object of type Building.
class BuildingDemo {
```

```
  public static void Main() {
    Building house = new Building(); // create a Building object
    int areaPP; // area per person

    // assign values to fields in house
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // compute the area per person
    areaPP = house.area / house.occupants;

    Console.WriteLine("house has:\n  " +
                      house.floors + " floors\n  " +
                      house.occupants + " occupants\n  " +
                      house.area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

This program consists of two classes: **Building** and **BuildingDemo**. Inside **BuildingDemo**, the **Main( )** method creates an instance of **Building** called **house**. Then the code within **Main( )** accesses the instance variables associated with **house**, assigning them values and using those values. It is important to understand that **Building** and **BuildingDemo** are two separate classes. The only relationship they have to each other is that one class creates an instance of the other. Although they are separate classes, code inside **BuildingDemo** can access the members of **Building** because they are declared **public**. If they had not been given the **public** access specifier, their access would have been limited to the **Building** class, and **BuildingDemo** would not have been able to use them.

Assume that you call the preceding file **UseBuilding.cs**. Compiling this program creates a file called **UseBuilding.exe**. Both the **Building** and **BuildingDemo** classes are automatically part of the executable file. The program displays the following output:

```
house has:
  2 floors
  4 occupants
```

```
2500 total area
625 area per person
```

Actually, it is not necessary for the **Building** and the **BuildingDemo** class to be in the same source file. You could put each class in its own file, called **Building.cs** and **BuildingDemo.cs**, for example. Just tell the C# compiler to compile both files and link them together. For example, you could use this command line to compile the program if you split it into two pieces as just described:

```
csc Building.cs BuildingDemo.cs
```

If you are using the Visual Studio IDE, you will need to add both files to your project and then build.

Before moving on, let's review a fundamental principle: each object has its own copies of the instance variables defined by its class. Thus, the contents of the variables in one object can differ from the contents of the variables in another. There is no connection between the two objects except for the fact that they are both objects of the same type. For example, if you have two **Building** objects, each has its own copy of **floors**, **area**, and **occupants**, and the contents of these can differ between the two objects. The following program demonstrates this fact:

```
// This program creates two Building objects.

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants
}

// This class declares two objects of type Building.
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();
```

```
    int areaPP; // area per person

    // assign values to fields in house
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // assign values to fields in office
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    // compute the area per person in house
    areaPP = house.area / house.occupants;

    Console.WriteLine("house has:\n  " +
                      house.floors + " floors\n  " +
                      house.occupants + " occupants\n  " +
                      house.area + " total area\n  " +
                            areaPP + " area per person");

    Console.WriteLine();

    // compute the area per person in office
    areaPP = office.area / office.occupants;

    Console.WriteLine("office has:\n  " +
                      office.floors + " floors\n  " +
                      office.occupants + " occupants\n  " +
                      office.area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

The output produced by this program is shown here:

```
house has:
  2 floors
  4 occupants
  2500 total area
```

625 area per person

office has:
3 floors
25 occupants
4200 total area
168 area per person

As you can see, **house**'s data is completely separate from the data contained in **office**. Figure 2-1 depicts this situation.

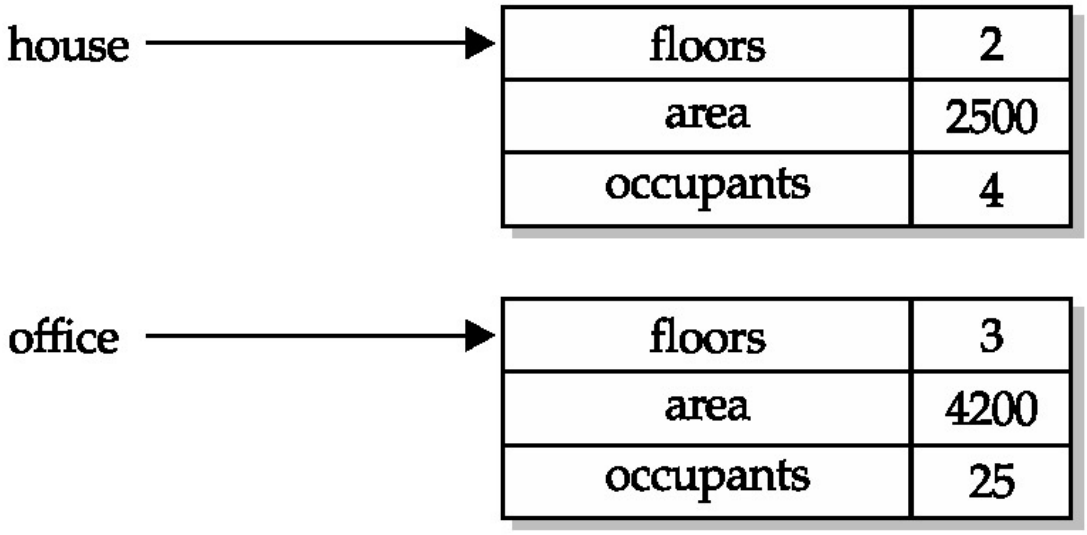


Figure 2-1: One object's instance variables are separate from another's.

How Objects are Created

In the preceding programs, the following line was used to declare an object of type **Building**:

```
Building house = new Building();
```

This declaration performs two functions. First, it declares a variable called **house** of the class type **Building**. This variable does not define an object. Instead, it is simply a variable that can *refer to* an object. Second, the declaration creates an actual, physical copy of the object and assigns to

house a reference to that object. This is done by using the **new** operator. Thus, after the line executes, **house** refers to an object of type **Building**.

The **new** operator dynamically allocates (that is, allocates at runtime) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in a variable. Thus, in C#, all class objects must be dynamically allocated.

The two steps combined in the preceding statement can be rewritten like this to show each step individually:

```
Building house; // declare reference to object
house = new Building(); // allocate a Building object
```

The first line declares **house** as a reference to an object of type **Building**. Thus, **house** is a variable that can refer to an object, but it is not an object, itself. The next line creates a new **Building** object and assigns a reference to it to **house**. Now, **house** is linked with an object.

The fact that class objects are accessed through a reference explains why classes are called *reference types.* The key difference between value types and reference types is what a variable of each type means. For a variable of a value type, the variable, itself, contains the value. For example, given

```
int x;
x = 10;
```

x contains the value 10 because **x** is a variable of type **int**, which is a value type. However, in the case of

```
Building house = new Building();
```

house does not, itself, contain the object. Instead, it contains a reference to the object

Reference Variables and Assignment

In an assignment operation, reference variables act differently than do variables of a value type, such as **int**. When you assign one value type variable to another, the situation is straightforward. The variable on the left

receives a *copy* of the *value* of the variable on the right. When you assign an object reference variable to another, the situation is a bit more complicated because you are changing the object to which the reference variable refers. The effect of this difference can cause some counterintuitive results. For example, consider the following fragment:

```
Building house1 = new Building();
Building house2 = house1;
```

At first glance, it is easy to think that **house1** and **house2** refer to different objects, but this is not the case. Instead, **house1** and **house2** will both refer to the *same* object. The assignment of **house1** to **house2** simply makes **house2** refer to the same object as does **house1**. Thus, the object can be acted upon by either **house1** or **house2**. For example, after the assignment

```
house1.area = 2600;
```

executes, both of these **WriteLine( )** statements

```
Console.WriteLine(house1.area);
Console.WriteLine(house2.area);
```

display the same value: 2600.

Although **house1** and **house2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **house2** simply changes the object to which **house2** refers. For example:

```
Building house1 = new Building();
Building house2 = house1;
Building house3 = new Building();

house2 = house3; // now house2 and house3 refer to the same object.
```

After this sequence executes, **house2** refers to the same object as **house3**. The object referred to by **house1** is unchanged

Methods

As explained, instance variables and methods are two of the primary constituents of classes. So far, the **Building** class contains data, but no

methods. Although data-only classes are perfectly valid, most classes will have methods. *Methods* are subroutines that manipulate the data defined by the class and, in many cases, provide access to that data. Typically, other parts of your program will interact with a class through its methods.

A method contains one or more statements. In well-written C# code, each method performs only one task. Each method has a name, and it is this name that is used to call the method. In general, you can give a method whatever name you please. However, remember that **Main( )** is reserved for the method that begins execution of your program. Also, don't use C#'s keywords for method names.

When denoting methods in text, this book has used and will continue to use a convention that has become common when writing about C#. A method will have parentheses after its name. For example, if a method's name is **getval**, then it will be written **getval( )** when its name is used in a sentence. This notation will help you distinguish variable names from method names in this book.

The general form of a method is shown here:

```
access ret-type name(parameter-list) {
  // body of method
}
```

Here, *access* is an access modifier that governs which other parts of your program can call the method. As explained earlier, the access modifier is optional. If not present, then the method is private to the class in which it is declared. For now, we will declare all methods as **public** so that they can be called by any other code in the program. The *ret-type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Adding a Method to the Building Class

As just explained, the methods of a class typically manipulate and provide access to the data of the class. With this in mind, recall that **Main( )** in the preceding examples computed the area-per-person by dividing the total area by the number of occupants. While technically correct, this is not the best way to handle this computation. The calculation of area-per-person is something that is best handled by the **Building** class, itself. The reason for this conclusion is easy to understand: The area-per-person of a building is dependent upon the values in the **area** and **occupants** fields, which are encapsulated by **Building**. Thus, it is possible for the **Building** class to perform this calculation on its own. Furthermore, by adding this calculation to **Building**, you prevent each program that uses **Building** from having to perform this calculation manually. This prevents the unnecessary duplication of code. Finally, by adding a method to **Building** that computes the area-per-person, you are enhancing its object-oriented structure by encapsulating the quantities that relate directly to a building inside **Building**.

To add a method to **Building**, specify it within **Building**'s declaration. For example, the following version of **Building** contains a method called **areaPerPerson( )** that displays the area-per-person for a building:

```
// Add a method to Building.

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants

  // Display the area per person.
  public void areaPerPerson() {
    Console.WriteLine("  " + area / occupants +
                      " area per person");
  }
}

// Use the areaPerPerson() method.
```

```
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();

    // assign values to fields in house
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // assign values to fields in office
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;
    Console.WriteLine("house has:\n  " +
                      house.floors + " floors\n  " +
                      house.occupants + " occupants\n  " +
                      house.area + " total area");
    house.areaPerPerson();

    Console.WriteLine();

    Console.WriteLine("office has:\n  " +
                      office.floors + " floors\n  " +
                      office.occupants + " occupants\n  " +
                      office.area + " total area");
    office.areaPerPerson();
  }
}
```

This program generates the following output, which is the same as before:

```
house has:
  2 floors
  4 occupants
  2500 total area
  625 area per person

office has:
  3 floors
```

```
25 occupants
4200 total area
168 area per person
```

Let's look at the key elements of this program, beginning with the **areaPerPerson( )** method, itself. The first line of **areaPerPerson( )** is

```
public void areaPerPerson() {
```

This line declares a method called **areaPerPerson** that has no parameters. It is specified as **public**, so it can be used by all other parts of the program. Its return type is **void**. Thus, **areaPerPerson( )** does not return a value to the caller. The line ends with the opening curly brace of the method body.

The body of **areaPerPerson( )** consists solely of this statement:

```
Console.WriteLine("  " + area / occupants +
                  " area per person");
```

This statement displays the area-per-person of a building by dividing **area** by **occupants**. Since each object of type **Building** has its own copy of **area** and **occupants**, when **areaPerPerson( )** is called, the computation uses the calling object's copies of those variables.

The **areaPerPerson( )** method ends when its closing curly brace is encountered. This causes program control to transfer back to the caller.

Next, look closely at this line of code from inside **Main( )**:

```
house.areaPerPerson();
```

This statement invokes the **areaPerPerson( )** method on **house**. That is, it calls **areaPerPerson( )** relative to the **house** object, using the object's name followed by the dot operator. When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call.

In this case, the call to **house.areaPerPerson( )** displays the area-per-person of the building defined by **house**. In similar fashion, the call to **office.areaPerPerson( )** displays the area-per-person of the building defined

by **office**. Each time **areaPerPerson( )** is invoked, it displays the area-per-person for the specified object.

There is something very important to notice inside the **areaPerPerson( )** method: The instance variables **area** and **occupants** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **area** and **occupants** inside **areaPerPerson( )** implicitly refer to the copies of those variables found in the object that invokes **areaPerPerson( )**.

Returning from a Method

In general, there are two conditions that cause a method to return. The first, as the **areaPerPerson( )** method in the preceding example shows, is when the method's closing curly brace is encountered. The second is when a **return** statement is executed. There are two forms of **return**: one for use in **void** methods (those that do not return a value) and one for returning values. The first form is examined here. The next section explains how to return values.

In a **void** method, you can cause the immediate termination of a method by using this form of **return**:

return ;

When this statement executes, program control returns to the caller, skipping any remaining code in the method. For example, consider this method:

```
public void myMeth() {
  int i;

  for(i=0; i<10; i++) {
    if(i == 5) return; // stop at 5
```

```
    Console.WriteLine();
  }
}
```

Here, the **for** loop will only run from 0 to 5, because once **i** equals 5, the method returns.

It is permissible to have multiple **return** statements in a method, especially when there are two or more routes out of it. For example,

```
public void myMeth() {
  // ...
  if(done) return;
  // ...
  if(error) return;
}
```

Here, the method returns if it is done or if an error occurs. Be careful, however. Having too many exit points in a method can destructure your code, so avoid using them casually.

To review: A **void** method can return in one of two ways—its closing curly brace is reached, or a **return** statement is executed.

Returning a Value

Although methods with a return type of **void** are not rare, most methods will return a value. In fact, the ability to return a value is one of the most useful features of a method.

Return values are used for a variety of purposes in programming. In some cases, the return value contains the outcome of some calculation. In other cases, the return value may simply indicate success or failure. In still others, it may contain a status code. Whatever the purpose, using method return values is an integral part of C# programming.

Methods return a value to the calling routine using this form of **return**:

return *value*;

Here, *value* is the value returned.

You can use a return value to improve the implementation of **areaPerPerson( )**. Instead of displaying the area-per-person, a better approach is to have **areaPerPerson( )** return this value. Among the advantages to this approach is that you can use the value for other calculations. The following example modifies **areaPerPerson( )** to return the area-per-person rather than displaying it:

```
// Return a value from areaPerPerson().

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants

  // Return the area per person.
  public int areaPerPerson() {
    return area / occupants;
  }
}

// Use the return value from areaPerPerson().
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();
    int areaPP; // area per person

    // assign values to fields in house
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // assign values to fields in office
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    // obtain area per person for house
```

```
    areaPP = house.areaPerPerson();

    Console.WriteLine("house has:\n  " +
                      house.floors + " floors\n  " +
                      house.occupants + " occupants\n  " +
                      house.area + " total area\n  " +
                      areaPP + " area per person");


    Console.WriteLine();

    // obtain area per person for office
    areaPP = office.areaPerPerson();

    Console.WriteLine("office has:\n  " +
                      office.floors + " floors\n  " +
                      office.occupants + " occupants\n  " +
                      office.area + " total area\n  " +
                      areaPP + " area per person");
  }
}
```

The output is the same as shown earlier.

In the program, notice that when **areaPerPerson( )** is called, it is put on the right side of an assignment statement. On the left is a variable that will receive the value returned by **areaPerPerson( )**. Thus, after

areaPP = house.areaPerPerson();

executes, the area-per-person of the **house** object is stored in **areaPP**.

Notice that **areaPerPerson( )** now has a return type of **int**. This means that it will return an integer value to the caller. The return type of a method is important because the type of data returned by a method must be compatible with the return type specified by the method. Thus, if you want a method to return data of type **double**, then its return type must be type **double**.

Although the preceding program is correct, it is not written as efficiently as it could be. Specifically, there is no need for the **areaPP** variable. A call to

areaPerPerson( ) can be used in the **WriteLine( )** statement directly, as shown here:

```
Console.WriteLine("house has:\n  " +
                  house.floors + " floors\n  " +
                  house.occupants + " occupants\n  " +
                  house.area + " total area\n  " +
                  house.areaPerPerson() + " area per person");
```

In this case, when **WriteLine( )** is executed, **house.areaPerPerson( )** is called automatically, and its value will be passed to **WriteLine( )**. Furthermore, you can use a call to **areaPerPerson( )** whenever the area-per-person of a **Building** object is needed. For example, this statement compares the per-person areas of two buildings:

```
if(b1.areaPerPerson() > b2.areaPerPerson())
  Console.WriteLine("b1 has more space for each person");
```

Lab 5: Introducing Classes and Objects

Part 2:

Using Parameters

It is possible to pass one or more values to a method when the method is called. As explained, a value passed to a method is called an *argument.* Inside the method, the variable that receives the argument is called a *parameter.* Parameters are declared inside the parentheses that follow the method's name. The parameter declaration syntax is the same as that used for variables. A parameter is within the scope of its method, and aside from its special task of receiving an argument, it acts like any other local variable.

Here is a simple example that uses a parameter. Inside the **ChkNum** class, the method **isPrime( )** returns **true** if the value that it is passed is prime. It returns **false** otherwise. Therefore, **isPrime( )** has a return type of **bool**.

```
// A simple example that uses a parameter.

using System;

class ChkNum {
  // Return true if x is prime.
  public bool isPrime(int x) {
    for(int i=2; i <= x/i; i++)
      if((x %i) == 0) return false;

    return true;
  }
}

class ParmDemo {
  public static void Main() {
    ChkNum ob = new ChkNum();

    for(int i=1; i < 10; i++)
      if(ob.isPrime(i)) Console.WriteLine(i + " is prime.");
      else Console.WriteLine(i + " is not prime.");
```

```
  }
}
```

Here is the output produced by the program:

```
1 is prime.
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.
```

In the program, **isPrime( )** is called nine times, and each time a different value is passed. Let's look at this process closely. First, notice how **isPrime( )** is called. The argument is specified between the parentheses. When **isPrime( )** is called the first time, it is passed value 1. Thus, when **isPrime( )** begins executing, the parameter **x** receives the value 1. In the second call, 2 is the argument, and **x** then has the value 2. In the third call, the argument is 3, which is the value that **x** receives, and so on. The point is that the value passed as an argument when **isPrime( )** is called is the value received by its parameter, **x**.

A method can have more than one parameter. Simply declare each parameter, separating one from the next with a comma. For example, here the **ChkNum** class is expanded by adding a method called **leastComFactor( )**, which returns the smallest factor that its two arguments have in common. In other words, it returns the smallest whole number value that can evenly divide both arguments.

```
// Add a method that takes two arguments.

using System;

class ChkNum {
  // Return true if x is prime.
  public bool isPrime(int x) {
    for(int i=2; i <= x/i; i++)
```

```
    if((x %i) == 0) return false;

   return true;
  }

  // Return the least common factor.
  public int leastComFactor(int a, int b) {
   int max;

   if(isPrime(a) | isPrime(b)) return 1;

   max = a < b ? a : b;

   for(int i=2; i <= max/2; i++)
    if(((a%i) == 0) & ((b%i) == 0)) return i;

   return 1;
  }
}

class ParmDemo {
  public static void Main() {
   ChkNum ob = new ChkNum();
   int a, b;

   for(int i=1; i < 10; i++)
    if(ob.isPrime(i)) Console.WriteLine(i + " is prime.");
    else Console.WriteLine(i + " is not prime.");

   a = 7;
   b = 8;
   Console.WriteLine("Least common factor for " +
                a + " and " + b + " is " +
                ob.leastComFactor(a, b));

   a = 100;
   b = 8;
   Console.WriteLine("Least common factor for " +
                a + " and " + b + " is " +
                ob.leastComFactor(a, b));
```

```
    a = 100;
    b = 75;
    Console.WriteLine("Least common factor for " +
                      a + " and " + b + " is " +
                      ob.leastComFactor(a, b));

  }
}
```

Notice that when **leastComFactor( )** is called, the arguments are also separated by commas. The output from the program is shown here:

```
1 is prime.
2 is prime.
3 is prime.
4 is not prime.
5 is prime.
6 is not prime.
7 is prime.
8 is not prime.
9 is not prime.
Least common factor for 7 and 8 is 1
Least common factor for 100 and 8 is 2
Least common factor for 100 and 75 is 5
```

When using multiple parameters, each parameter specifies its own type, which can differ from the others. For example, this is perfectly valid:

```
int myMeth(int a, double b, float c) {
 // ...
```

Adding a Parameterized Method to Building

You can use a parameterized method to add a new feature to the **Building** class: the ability to compute the maximum number of occupants for a building assuming that each occupant must have a certain minimal space. This new method is called **maxOccupant( )**. It is shown here:

```
/* Return the maximum number of occupants if each
   is to have at least the specified minimum area. */
public int maxOccupant(int minArea) {
```

```
  return area / minArea;
}
```

When **maxOccupant( )** is called, the parameter **minArea** receives the minimum space needed for each occupant. The method divides the total area of the building by this value and returns the result.

The entire **Building** class that includes **maxOccupant( )** is shown here:

```
/*
   Add a parameterized method that computes the
   maximum number of people that can occupy a
   building assuming each needs a specified
   minimum space.
*/

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants

  // Return the area per person.
  public int areaPerPerson() {
    return area / occupants;
  }

  /* Return the maximum number of occupants if each
     is to have at least the specified minimum area. */
  public int maxOccupant(int minArea) {
    return area / minArea;
  }
}

// Use maxOccupant().
class BuildingDemo {
  public static void Main() {
    Building house = new Building();
    Building office = new Building();
```

```
    // assign values to fields in house
    house.occupants = 4;
    house.area = 2500;
    house.floors = 2;

    // assign values to fields in office
    office.occupants = 25;
    office.area = 4200;
    office.floors = 3;

    Console.WriteLine("Maximum occupants for house if each has " +
                      300 + " square feet: " +
                      house.maxOccupant(300));

    Console.WriteLine("Maximum occupants for office if each has " +
                      300 + " square feet: " +
                      office.maxOccupant(300));
  }
}
```

The output from the program is shown here:

```
Maximum occupants for house if each has 300 square feet: 8
Maximum occupants for office if each has 300 square feet: 14
```

Avoiding Unreachable Code

When creating methods, you should avoid causing a situation in which a portion of code cannot, under any circumstances, be executed. This is called *unreachable code*, and it is considered incorrect in C#. The compiler will issue a warning message if you create a method that contains unreachable code. For example:

```
public void m() {
  char a, b;

  // ...
```

```
  if(a==b) {
    Console.WriteLine("equal");
    return;
  } else {
    Console.WriteLine("not equal");
    return;
  }
  Console.WriteLine("this is unreachable");
}
```

Here, the method **m( )** will always return before the final **WriteLine( )** statement is executed. If you try to compile this method, you will receive a warning. In general, unreachable code constitutes a mistake on your part, so it is a good idea to take unreachable code warnings seriously.

Constructors

In the preceding examples, the instance variables of each **Building** object had to be set manually using a sequence of statements such as:

```
house.occupants = 4;
house.area = 2500;
house.floors = 2;
```

An approach like this would never be used in professionally written C# code. Aside from this approach being error prone (you might forget to set one of the fields), there is simply a better way to accomplish this task: the constructor.

A *constructor* initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. The general form of constructor is shown here:

```
access class-name( ) {
  // constructor code
}
```

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object. Also, usually, *access* is **public** because constructors are normally called from outside their class.

All classes have constructors, whether you define one or not, because C# automatically provides a default constructor that initializes all member variables to zero (for value types) or null (for reference types). However, once you define your own constructor, the default constructor is no longer used.

Here is a simple example that uses a constructor:

```
// A simple constructor.

using System;

class MyClass {
  public int x;

  public MyClass() {
    x = 10;
  }
}

class ConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

In this example, the constructor for **MyClass** is

```
public MyClass() {
  x = 10;
}
```

Notice that the constructor is specified as **public**. This is because the constructor will be called from code defined outside of its class. This constructor assigns the instance variable **x** of **MyClass** the value 10. This constructor is called by **new** when an object is created. For example, in the line

MyClass t1 = new MyClass();

the constructor **MyClass( )** is called on the **t1** object, giving **t1.x** the value 10. The same is true for **t2**. After construction, **t2.x** has the value 10. Thus, the output from the program is

10 10

Parameterized Constructors

In the preceding example, a parameterless constructor was used. While this is fine for some situations, most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name. For example, here **MyClass** is given a parameterized constructor:

```
// A parameterized constructor.

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

The output from this program is shown here:

10 88

In this version of the program, the **MyClass( )** constructor defines one parameter called **i**, which is used to initialize the instance variable, **x**. Thus, when the line

MyClass t1 = new MyClass(10);

executes, the value 10 is passed to **i**, which is then assigned to **x**.

Adding a Constructor to the Building Class

We can improve the **Building** class by adding a constructor that automatically initializes the **floors**, **area**, and **occupants** fields when an object is constructed. Pay special attention to how **Building** objects are created.

```
// Add a constructor to Building.

using System;

class Building {
  public int floors;    // number of floors
  public int area;      // total square footage of building
  public int occupants; // number of occupants

  public Building(int f, int a, int o) {
    floors = f;
    area = a;
    occupants = o;
  }

  // Display the area per person.
  public int areaPerPerson() {
    return area / occupants;
  }

  /* Return the maximum number of occupants if each
     is to have at least the specified minimum area. */
```

```
  public int maxOccupant(int minArea) {
    return area / minArea;
  }
}

// Use the parameterized Building constructor.
class BuildingDemo {
  public static void Main() {
    Building house = new Building(2, 2500, 4);
    Building office = new Building(3, 4200, 25);

    Console.WriteLine("Maximum occupants for house if each has " +
                      300 + " square feet: " +
                      house.maxOccupant(300));

    Console.WriteLine("Maximum occupants for office if each has " +
                      300 + " square feet: " +
                      office.maxOccupant(300));
  }
}
```

The output from this program is the same as for the previous version.

Both **house** and **office** were initialized by the **Building( )** constructor when they were created. Each object is initialized as specified in the parameters to its constructor. For example, in the following line,

Building house = new Building(2, 2500, 4);

the values 2, 2500, and 4 are passed to the **Building( )** constructor when **new** creates the object. Thus, **house**'s copy of **floors**, **area**, and **occupants** will contain the values 2, 2500, and 4, respectively.

The new Operator Revisited

Now that you know more about classes and their constructors, let's take a closer look at the **new** operator. As it relates to classes, the **new** operator has this general form:

*class-var = new cla*ss-name( );

Here, *class-var* is a variable of the class type being created. The *class-name* is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. If a class does not define its own constructor, **new** will use the default constructor supplied by C#. Thus, **new** can be used to create an object of any class type.

Since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a runtime exception will occur.

Using new with Value Types

At this point, you might be asking why you don't need to use **new** for variables of the value types, such as **int** or **float**? In C#, a variable of a value type contains its own value. Memory to hold this value is automatically allocated by the compiler when a program is compiled. Thus, there is no need to explicitly allocate this memory using **new**. Conversely, a reference variable stores a reference to an object. The memory to hold this object is allocated dynamically, during execution.

Not making the fundamental types, such as **int** or **char**, into reference types greatly improves the performance of your program. When using a reference type, there is a layer of indirection that adds overhead to each object access that is avoided by a value type.

As a point of interest, it is permitted to use **new** with the value types, as shown here:

```
int i = new int();
```

Doing so invokes the default constructor for type **int**, which initializes **i** to zero. For example:

```
// Use new with a value type.

using System;

class newValue {
  public static void Main() {
    int i = new int(); // initialize i to zero
```

```
    Console.WriteLine("The value of i is: " + i);
  }
}
```

The output from this program is

The value of i is: 0

As the output verifies, **i** is initialized to zero. Remember, without the use of **new**, **i** would be uninitialized, and it would be an error to attempt to use it in the **WriteLine( )** statement without explicitly giving it a value.

In general, invoking **new** for a value type invokes the default constructor for that type. It does not, however, dynamically allocate memory. Frankly, most programmers do not use **new** with the value types.

Garbage Collection and Destructors

As you have seen, objects are dynamically allocated from a pool of free memory by using the **new** operator. Of course, memory is not infinite, and the free memory can be exhausted. Thus, it is possible for **new** to fail because there is insufficient free memory to create the desired object. For this reason, one of the key components of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation. In many programming languages, the release of previously allocated memory is handled manually. For example, in C++, you use the **delete** operator to free memory that was allocated. However, C# uses a different, more trouble-free approach: *garbage collection.*

C#'s garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention. It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released. This recycled memory can then be used for a subsequent allocation.

Garbage collection occurs only sporadically during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Thus, you can't know, or make assumptions about, precisely when garbage collection will take place.

Destructors

It is possible to define a method that will be called prior to an object's final destruction by the garbage collector. This method is called a *destructor,* and it can be used to ensure that an object terminates cleanly. For example, you might use a destructor to make sure that an open file owned by that object is closed.

Destructors have this general form:

```
~class-name( ) {
  // destruction code
}
```

Here, *class-name* is the name of the class. Thus, a destructor is declared like a constructor except that it is preceded with a ~ (tilde). Notice it has no return type.

To add a destructor to a class, you simply include it as a member. It is called whenever an object of its class is about to be recycled. Inside the destructor, you will specify those actions that must be performed before an object is destroyed.

It is important to understand that the destructor is called prior to garbage collection. It is not called when an object goes out of scope, for example. (This differs from destructors in C++, which *are* called when an object goes out of scope.) This means that you cannot know precisely when a destructor will be executed. However, all destructors will be called before a program terminates.

The following program demonstrates a destructor. It works by creating and destroying a large number of objects. During this process, at some point the garbage collector will be activated, and the destructors for the objects will be called.

```
// Demonstrate a destructor.

using System;

class Destruct {
  public int x;
```

```
  public Destruct(int i) {
    x = i;
  }

  // called when object is recycled
  ~Destruct() {
    Console.WriteLine("Destructing " + x);
  }

  // generates an object that is immediately destroyed
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }

}

class DestructDemo {
  public static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* Now, generate a large number of objects.  At
       some point, garbage collection will occur.
       Note: you might need to increase the number
       of objects generated in order to force
       garbage collection. */
    for(count=1; count < 100000; count++)
      ob.generator(count);

    Console.WriteLine("Done");
  }
}
```

Here is how the program works. The constructor sets the instance variable **x** to a known value. In this example, **x** is used as an object ID. The destructor displays the value of **x** when an object is recycled. Of special interest is **generator( )**. This method creates and then promptly discards a **Destruct** object (because **o** goes out of scope when **generator( )** returns). The

DestructDemo class creates an initial **Destruct** object called **ob**. Then using **ob**, it creates 100,000 objects by calling **generator( )** on **ob**. This has the net effect of creating and discarding 100,000 objects. At various points in the middle of this process, garbage collection will take place. Precisely how often or when is dependent upon several factors, such as the initial amount of free memory, the operating system, and so on. However, at some point, you will start to see the messages generated by the destructor. If you don't see the messages prior to program termination (that is, before you see the "Done" message), try increasing the number of objects being generated by upping the count in the **for** loop.

Because of the nondeterministic way in which destructors are called, they should not be used to perform actions that must occur at a specific point in your program. One other point: it is possible to request garbage collection. However, manually initiating garbage collection is not recommended for most circumstances, because it can lead to inefficiencies. Also, because of the way the garbage collector works, even if you explicitly request garbage collection, there is no way to know precisely when a specific object will be recycled.

The this Keyword

When a method is called, it is automatically passed an implicit argument that is a reference to the invoking object (that is, the object on which the method is called). This reference is called **this**. To understand **this**, first consider a program that creates a class called **Rect** that encapsulates the width and height of a rectangle and that includes a method called **area( )** that returns its area:

```
using System;

class Rect {
  public int width;
  public int height;

  public Rect(int w, int h) {
    width = w;
    height = h;
  }
```

```
  public int area() {
    return width * height;
  }
}

class UseRect {
  public static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Area of r1: " + r1.area());

    Console.WriteLine("Area of r2: " + r2.area());

  }
}
```

As you know, within a method, the other members of a class can be accessed directly, without any object or class qualification. Thus, inside **area( )**, the statement

```
return width * height;
```

means that the copies of **width** and **height** associated with the invoking object will be multiplied together and the result returned. However, the same statement can also be written like this:

```
return this.width * this.height;
```

Here, **this** refers to the object on which **area( )** was called. Thus, **this.width** refers to that object's copy of **width**, and **this.height** refers to that object's copy of **height**. For example, if **area( )** had been invoked on an object called **x**, then **this** in the preceding statement would have been referring to **x**. Writing the statement without using **this** is really just shorthand.

Here is the entire **Rect** class written using the **this** reference:

```
using System;

class Rect {
  public int width;
```

```
  public int height;

  public Rect(int w, int h) {
    this.width = w;
    this.height = h;
  }

  public int area() {
    return this.width * this.height;
  }
}

class UseRect {
  public static void Main() {
    Rect r1 = new Rect(4, 5);
    Rect r2 = new Rect(7, 9);

    Console.WriteLine("Area of r1: " + r1.area());

    Console.WriteLine("Area of r2: " + r2.area());

  }
}
```

Actually, no C# programmer would use **this** as just shown, because nothing is gained, and the standard form is easier. However, **this** has some important uses. For example, the C# syntax permits the name of a parameter or a local variable to be the same as the name of an instance variable. When this happens, the local name *hides* the instance variable. You can gain access to the hidden instance variable by referring to it through **this**. For example, while not recommended style, the following is a syntactically valid way to write the **Rect( )** constructor:

```
public Rect(int width, int height) {
  this.width = width;
  this.height = height;
}
```

In this version, the names of the parameters are the same as the names of the instance variables, thus hiding them. However, **this** is used to “uncover” the instance variables.

Lab 6: Inheritance

Inheritance is one of the three foundational principles of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits (خواص) common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

In the language of C#, a class that is inherited is called a *base class*. The class that does the inheriting is called a *derived class*. Therefore, a derived class is a specialized version of a base class. It inherits all of the variables, methods, properties, operators, and indexers defined by the base class and adds its own unique elements.

Inheritance Basics

C# supports inheritance by allowing one class to incorporate another class into its declaration. This is done by specifying a base class when a derived class is declared. Let's begin with an example. The following class called **TwoDShape** defines the attributes of a "generic" two-dimensional shape, such as a square, rectangle, triangle, and so on.

```
// A class for two-dimensional objects.
class TwoDShape {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Width and height are " +
                      width + " and " + height);
  }
}
```

TwoDShape can be used as a base class (that is, as a starting point) for classes that describe specific types of two-dimensional objects. For example, the following program uses **TwoDShape** to derive a class called **Triangle**. Pay close attention to the way that **Triangle** is declared.

```
// A simple class hierarchy.

using System;
// A class for two-dimensional objects.
```

```
class TwoDShape {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Width and height are " +
                      width + " and " + height);
  }
}

// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
  public string style; // style of triangle

  // Return area of triangle.
  public double area() {
    return width * height / 2;
  }

  // Display a triangle's style.
  public void showStyle() {
    Console.WriteLine("Triangle is " + style);
  }
}

class Shapes {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();


    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "isosceles";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "right";

    Console.WriteLine("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Area is " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
```

```
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Area is " + t2.area());
  }
}
```

The **Triangle** class creates a specific type of **TwoDShape**, in this case, a triangle. The **Triangle** class includes all of **TwoDShape** and adds the field **style**, the method **area( )**, and the method **showStyle( )**. A description of the type of triangle is stored in **style**, **area( )** computes and returns the area of the triangle, and **showStyle( )** displays the triangle style.

Notice the syntax that **Triangle** uses to inherit **TwoDShape**:

```
class Triangle : TwoDShape {
```

This syntax can be generalized. Whenever one class inherits another, the base class name follows the name of the derived class, separated by a colon. In C#, the syntax for inheriting a class is remarkably simple and easy-to-use.

Because **Triangle** includes all of the members of its base class, **TwoDShape**, it can access **width** and **height** inside **area( )**. Also, inside **Main( )**, objects **t1** and **t2** can refer to **width** and **height** directly, as if they were part of **Triangle**. Figure 4-1 depicts conceptually how **TwoDShape** is incorporated into **Triangle**.

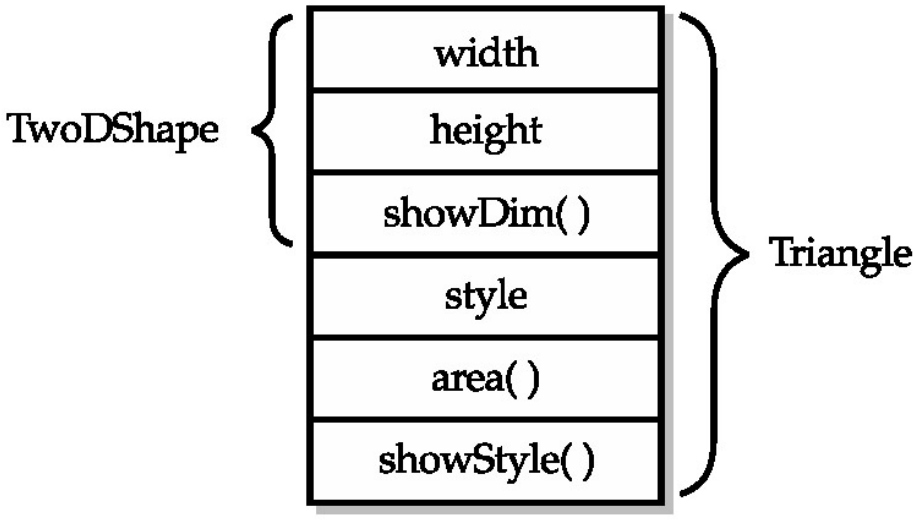


Figure 6-1: A conceptual depiction of the Triangle class

Even though **TwoDShape** is a base for **Triangle**, it is also a completely independent, stand-alone class. Being a base class for a derived class does not mean that the base class cannot be used by itself. For example, the following is perfectly valid:

```
TwoDShape shape = new TwoDShape();

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Of course, an object of **TwoDShape** has no knowledge of or access to any classes derived from **TwoDShape**.

The general form of a **class** declaration that inherits a base class is shown here:

```
class derived-class-name : base-class-name {
  // body of class
}
```

You can specify only one base class for any derived class that you create. C# does not support the inheritance of multiple base classes into a single derived class. (This differs from C++, in which you can inherit multiple base classes. Be aware of this when converting C++ code to C#.) You can, however, create a hierarchy of inheritance in which a derived class becomes a base class of another derived class. Of course, no class can be a base class of itself, either directly or indirectly.

A major advantage of inheritance is that once you have created a base class that defines the attributes common to a set of objects, it can be used to create any number of more specific derived classes. Each derived class can precisely tailor its own classification. For example, here is another class derived from **TwoDShape** that encapsulates rectangles:

```
// A derived class of TwoDShape for rectangles.
class Rectangle : TwoDShape {
  // Return true if the rectangle is square.
  public bool isSquare() {
```

```
    if(width == height) return true;
    return false;
  }

  // Return area of the rectangle.
  public double area() {
    return width * height;
  }
}
```

The **Rectangle** class includes **TwoDShape** and adds the methods **isSquare( )**, which determines if the rectangle is square, and **area( )**, which computes the area of a rectangle.

Member Access and Inheritance

members of a class are often declared private to prevent their unauthorized use or tampering. Inheriting a class *does not* overrule the private access restriction. Thus, even though a derived class includes all of the members of its base class, it cannot access those members of the base class that are private. For example, if, as shown here, **width** and **height** are made private in **TwoDShape**, then **Triangle** will not be able to access them:

```
// Private members are not inherited.

// This example will not compile.
using System;

// A class for two-dimensional objects.
class TwoDShape {
  double width;  // now private
  double height; // now private
  public void showDim() {
    Console.WriteLine("Width and height are " +
                      width + " and " + height);
  }
}

// Triangle is derived from TwoDShape.
class Triangle : TwoDShape {
  public string style; // style of triangle

  // Return area of triangle.
  public double area() {
```

```
    return width * height / 2; // Error, can't access
private member
  }

  // Display a triangle's style.
  public void showStyle() {
    Console.WriteLine("Triangle is " + style);
  }
}
```

The **Triangle** class will not compile because the reference to **width** and **height** inside the **area( )** method causes an access violation. Since **width** and **height** are now private, they are only accessible by other members of their own class. Derived classes have no access to them.

REMEMBER *A private class member will remain private to its class. It is not accessible by any code outside its class, including derived classes.*

At first, you might think that it is a serious restriction that derived classes do not have access to the private members of base classes, because it would prevent the use of private members in many situations. However, this is not true, because C# provides various solutions. One is to use **protected** members, which is described in the next section. A second is to use public properties to provide access to private data. As you have seen in the preceding labs, C# programmers typically grant access to the private members of a class through methods or by making them into properties. Here is a rewrite of the **TwoDShape** classes that makes **width** and **height** into properties:

```
// Use properties to set and get private members.

using System;

// A class for two-dimensional objects.
class TwoDShape {
  double pri_width;  // now private
  double pri_height; // now private

  // Properties for width and height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }
```

```
  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Width and height are " +
                      width + " and " + height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  public string style; // style of triangle

  // Return area of triangle.
  public double area() {
    return width * height / 2;
  }

  // Display a triangle's style.
  public void showStyle() {
    Console.WriteLine("Triangle is " + style);
  }
}

class Shapes2 {
  public static void Main() {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "isosceles";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "right";

    Console.WriteLine("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Area is " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
```

```
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Area is " + t2.area());
  }
}
```

In this version, the properties **width** and **height** provide access to the private members, **pri_width** and **pri_height**, which actually store the values. Therefore, even though **pri_width** and **pri_height** are private to **TwoDShape**, their values can still be set and obtained through their corresponding public properties.

When referring to base and derived classes, sometimes the terms *superclass* and *subclass* are used. These terms come from Java programming. What Java calls a superclass, C# calls a base class. What Java calls a subclass, C# calls a derived class. You will commonly hear both sets of terms applied to a class of either language, but this lab will continue to use the standard C# terms. C++ also uses the base-class/derived-class terminology.

Using Protected Access

As just explained, a private member of a base class is not accessible by a derived class. This would seem to imply that if you wanted a derived class to have access to some member in the base class, it would need to be public. Of course, making the member public also makes it available to all other code, which may not be desirable. Fortunately, this implication is untrue because C# allows you to create a *protected member.* A protected member is public within a class hierarchy, but private outside that hierarchy.

A protected member is created by using the **protected** access modifier. When a member of a class is declared as **protected**, that member is, with one important exception, private. The exception occurs when a protected member is inherited. In this case, a protected member of the base class becomes a protected member of the derived class and is, therefore, accessible by the derived class. Thus, by using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

Here is a simple example that uses **protected**:

```
// Demonstrate protected.
```

```
using System;

class B {
  protected int i, j; // private to B, but accessible by D

  public void set(int a, int b) {
    i = a;
    j = b;
  }

  public void show() {
    Console.WriteLine(i + " " + j);
 }
}

class D : B {
  int k; // private

  // D can access B's i and j
  public void setk() {
     k = i * j;
  }

  public void showk() {
    Console.WriteLine(k);
  }
}

class ProtectedDemo {
  public static void Main() {
    D ob = new D();

    ob.set(2, 3); // OK, known to D
    ob.show();    // OK, known to D

    ob.setk();  // OK, part of D
    ob.showk(); // OK, part of D
  }
}
```

In this example, because **B** is inherited by **D** and because **i** and **j** are declared as **protected** in **B**, the **setk( )** method can access them. If **i** and **j** had been declared as private by **B**, then **D** would not have access to them, and the program would not compile.

Like **public** and **private**, **protected** status stays with a member no matter how many layers of inheritance are involved. Therefore, when a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited by the first derived class is also inherited as protected by a second derived class.

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies consisting of only a base class and a derived class. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a derived class as a base class of another. For example, given three classes called **A**, **B**, and **C**, **C** can be derived from **B**, which can be derived from **A**. When this type of situation occurs, each derived class inherits all of the traits found in all of its base classes. In this case, **C** inherits all aspects of **B** and **A**.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the derived class **Triangle** is used as a base class to create the derived class called **ColorTriangle**. **ColorTriangle** inherits all of the traits of **Triangle** and **TwoDShape**, and adds a field called **color**, which holds the color of the triangle.

```
// A multilevel hierarchy.

using System;

class TwoDShape {
  double pri_width;  // private
  double pri_height; // private

  // Default constructor.
  public TwoDShape() {
    width = height = 0.0;
  }

  // Constructor for TwoDShape.
  public TwoDShape(double w, double h) {
    width = w;
    height = h;
  }

  // Construct object with equal width and height.
```

```
  public TwoDShape(double x) {
    width = height = x;
  }

  // Properties for width and height.
  public double width {
     get { return pri_width; }
     set { pri_width = value; }
  }

  public double height {
     get { return pri_height; }
     set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Width and height are " +
                      width + " and " + height);
  }
}

// A derived class of TwoDShape for triangles.
class Triangle : TwoDShape {
  string style; // private

  /* A default constructor. This invokes the default
     constructor of TwoDShape. */
  public Triangle() {
    style = "null";
  }

  // Constructor
  public Triangle(string s, double w, double h) : base(w,
h) {
    style = s;
  }

  // Construct an isosceles triangle.
  public Triangle(double x) : base(x) {
    style = "isosceles";
  }

  // Return area of triangle.
  public double area() {
    return width * height / 2;
  }
```

```
  // Display a triangle's style.
  public void showStyle() {
    Console.WriteLine("Triangle is " + style);
  }
}

// Extend Triangle.
class ColorTriangle : Triangle {
  string color;

  public ColorTriangle(string c, string s,
                       double w, double h) : base(s, w, h)
{
    color = c;
  }

  // Display the color.
  public void showColor() {
    Console.WriteLine("Color is " + color);
  }
}

class Shapes6 {
  public static void Main() {
    ColorTriangle t1 =
         new ColorTriangle("Blue", "right", 8.0, 12.0);
    ColorTriangle t2 =
         new ColorTriangle("Red", "isosceles", 2.0, 2.0);

    Console.WriteLine("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    Console.WriteLine("Area is " + t1.area());

    Console.WriteLine();

    Console.WriteLine("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    Console.WriteLine("Area is " + t2.area());
  }
}
```

Because of inheritance, **ColorTriangle** can make use of the previously defined classes of **Triangle** and **TwoDShape**, adding only the extra

information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **base** always refers to the constructor in the closest base class. The **base** in **ColorTriangle** calls the constructor in **Triangle**. The **base** in **Triangle** calls the constructor in **TwoDShape**. In a class hierarchy, if a base class constructor requires parameters, then all derived classes must pass those parameters "up the line." This is true whether or not a derived class needs parameters of its own.

Lab 7: Interfaces

In object-oriented programming it is sometimes helpful to define what a class must do, but not how it will do it. An example of this: the abstract method. An abstract method declares the return type and signature for a method, but provides no implementation. A derived class must provide its own implementation of each abstract method defined by its base class. Thus, an abstract method specifies the *interface* to the method, but not the *implementation.* While abstract classes and methods are useful, it is possible to take this concept a step further. In C#, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces are syntactically similar to abstract classes. However, in an interface, no method can include a body. That is, an interface provides no implementation whatsoever. It specifies what must be done, but not how. Once an interface is defined, any number of classes can implement it. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide bodies (implementations) for the methods described by the interface. Each class is free to determine the details of its own implementation. Thus, two classes might implement the same interface in different ways, but each class still supports the same set of methods. Therefore, code that has knowledge of the interface can use objects of either class since the interface to those objects is the same. By providing the interface, C# allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are declared by using the **interface** keyword. Here is a simplified form of an interface declaration:

```
interface name {
  ret-type method-name1(param-list);
  ret-type method-name2(param-list);
  // ...
  ret-type method-nameN(param-list);
}
```

The name of the interface is specified by *name.* Methods are declared using only their return type and signature. They are, essentially, abstract methods. As explained, in an **interface**, no method can have an implementation. Thus, each class that includes an **interface** must implement all of the methods. In an interface, methods are implicitly **public**, and no explicit access specifier is allowed.

Here is an example of an **interface**. It specifies the interface to a class that generates a series of numbers.

```
public interface ISeries {
  int getNext(); // return next number in series
  void reset(); // restart
  void setStart(int x); // set starting value
}
```

The name of this interface is **ISeries**. Although the prefix **I** is not necessary, many programmers prefix interfaces with it to differentiate them from classes. **ISeries** is declared **public** so that it can be implemented by any class in any program.

In addition to methods, interfaces can specify properties, indexers, and events. We will be concerned with only methods, properties, and indexers here. Interfaces cannot have data members. They cannot define constructors, destructors, or operator methods. Also, no member can be declared as **static**.

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, the name of the interface is specified after the class name in just the same way that a base class is specified. The general form of a class that implements an interface is shown here:

```
class class-name : interface-name {
   // class-body
}
```

The name of the interface being implemented is specified in *interface-name.*

When a class implements an interface, the class must implement the entire interface. It cannot pick and choose which parts to implement, for example.

A class can implement more than one interface. When a class implements more than one interface, specify each interface in a comma-separated list. A class can inherit a base class and also implement one or more interfaces. In this case, the name of the base class must come first in the comma-separated list.

The methods that implement an interface must be declared **public**. The reason for this is that methods are implicitly public within an interface, so their implementations must also be public. Also, the return type and signature of the implementing method must match exactly the return type and signature specified in the **interface** definition.

Here is an example that implements the **ISeries** interface shown earlier. It creates a class called **ByTwos**, which generates a series of numbers, each two greater than the previous one.

```
// Implement ISeries.
class ByTwos : ISeries {
  int start;
  int val;

  public ByTwos() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
```

```
    start = x;
    val = start;
  }
}
```

As you can see, **ByTwos** implements all three methods defined by **ISeries**. As explained, this is necessary since a class cannot create a partial implementation of an interface.

Here is a class that demonstrates **ByTwos**:

```
// Demonstrate the ByTwos interface.

using System;

class SeriesDemo {
  public static void Main() {
    ByTwos ob = new ByTwos();

    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.getNext());

    Console.WriteLine("\nResetting");
    ob.reset();
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.getNext());

    Console.WriteLine("\nStarting at 100");
    ob.setStart(100);
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " +
                        ob.getNext());
  }
}
```

To compile **SeriesDemo**, you must include the files that contain **ISeries**, **ByTwos**, and **SeriesDemo** in the compilation. The compiler will automatically compile all three files to create the final executable. For

example, if you called these files **ISeries.cs**, **ByTwos.cs**, and **SeriesDemo.cs**, then the following command line will compile the program:

```
>csc SeriesDemo.cs ISeries.cs ByTwos.cs
```

If you are using the Visual Studio IDE, simply add all three files to your C# project. One other point: it is perfectly valid to put all three of these classes in the same file, too.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **ByTwos** adds the method **getPrevious( )**, which returns the previous value:

```
// Implement ISeries and add getPrevious().
 class ByTwos : ISeries {
  int start;
  int val;
  int prev;

  public ByTwos() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void reset() {
    val = start;
    prev = start - 2;
  }

  public void setStart(int x) {
    start = x;
```

```
    val = start;
    prev = val - 2;
  }

  // A method not specified by ISeries.
  public int getPrevious() {
    return prev;
  }
}
```

Notice that the addition of **getPrevious( )** required a change to implementations of the methods defined by **ISeries**. However, since the interface to those methods stays the same, the change is seamless and does not break preexisting code. This is one of the advantages of interfaces.

As explained, any number of classes can implement an **interface**. For example, here is a class called **Primes** that generates a series of prime numbers. Notice that its implementation of **ISeries** is fundamentally different than the one provided by **ByTwos**.

```
// Use ISeries to implement a series of prime numbers.
class Primes : ISeries {
  int start;
  int val;

  public Primes() {
    start = 2;
    val = 2;
  }

  public int getNext() {
    int i, j;
    bool isprime;

    val++;
    for(i = val; i < 1000000; i++) {
      isprime = true;
      for(j = 2; j < (i/j + 1); j++) {
        if((i%j)==0) {
          isprime = false;
```

```
          break;
        }
      }
      if(isprime) {
        val = i;
        break;
      }
    }
    return val;
  }

  public void reset() {
    val = start;
  }

  public void setStart(int x) {
    start = x;
    val = start;
  }
}
```

The key point is that even though **ByTwos** and **Primes** generate completely unrelated series of numbers, both implement **ISeries**. As explained, an interface says nothing about the implementation, so each class is free to implement the interface as it sees fit.

Interface Properties

Like methods, properties are specified in an interface without any body. Here is the general form of a property specification:

```
// interface property
type name {
  get;
  set;
}
```

Of course, only **get** or **set** will be present for read-only or write-only properties, respectively.

Here is a rewrite of the **ISeries** interface and the **ByTwos** class that uses a property to obtain and set the next element in the series:

```
// Use a property in an interface.

using System;

public interface ISeries {
  // An interface property.
  int next {
    get; // return the next number in series
    set; // set next number
  }
}

// Implement ISeries.
class ByTwos : ISeries {
  int val;

  public ByTwos() {
    val = 0;
  }

  // get or set value
  public int next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }
}

// Demonstrate an interface property.
class SeriesDemo3 {
  public static void Main() {
    ByTwos ob = new ByTwos();

    // access series through a property
```

```
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob.next);

    Console.WriteLine("\nStarting at 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
      Console.WriteLine("Next value is " + ob.next);
  }
}
```

Lab 8: Using I/O

FileStream and Byte-Oriented File I/O

C# provides classes that allow you to read and write files. Of course, the most common type of files are disk files. At the operating system level, all files are byte oriented. As you would expect, C# provides methods to read and write bytes from and to a file. Thus, reading and writing files using byte streams is very common. C# also allows you to wrap a byte-oriented file stream within a character-based object. Character-based file operations are useful when text is being stored. Character streams are discussed later in this chapter. Byte-oriented I/O is described here.

To create a byte-oriented stream attached to a file, you will use the **FileStream** class. **FileStream** is derived from **Stream** and contains all of **Stream**'s functionality.

Remember, the stream classes, including **FileStream**, are defined in **System.IO**. Thus, you will usually include

```
using System.IO;
```

near the top of any program that uses them.

Opening and Closing a File

To create a byte stream linked to a file, create a **FileStream** object. **FileStream** defines several constructors. Perhaps its most commonly used one is shown here:

```
FileStream(string filename, FileMode mode)
```

Here, *filename* specifies the name of the file to open, which can include a full path specification. The *mode* parameter specifies how the file will be opened. It must be one of the values defined by the **FileMode** enumeration. These values are shown in Table 9-1. This constructor opens a file for read/write access.

Table 8-1: The FileMode Values

Value	Description
FileMode.Append	Output is appended to the end of file.
FileMode.Create	Creates a new output file. Any preexisting file by the same name will be destroyed.
FileMode.CreateNew	Creates a new output file. The file must not already exist.
FileMode.Open	Opens a preexisting file.
FileMode.OpenOrCreate	Opens a file if it exists, or creates the file if it does not already exist.
FileMode.Truncate	Opens a preexisting file, but reduces its length to zero.

If a failure occurs when attempting to open the file, an exception will be thrown. If the file cannot be opened because it does not exist, **FileNotFoundException** will be thrown. If the file cannot be opened because of an I/O error, **IOException** will be thrown. Other possible exceptions are **ArgumentNullException** (the filename is null), **ArgumentException** (the filename is invalid), **SecurityException** (user does not have access rights), **DirectoryNotFoundException** (specified directory is invalid), **PathTooLongException** (the filename/path is too long), and **ArgumentOutOfRangeException** (*mode* is invalid).

The following shows one way to open the file **test.dat** for input:

```
FileStream fin;

try {
  fin = new FileStream("test.dat", FileMode.Open);
}
catch(FileNotFoundException exc) {
  Console.WriteLine(exc.Message);
  return;
}
catch {
  Console.WriteLine("Cannot open file.");
  return;
}
```

Here, the first **catch** clause catches the file-not-found error. The second **catch**, which is a "catch all" clause, handles the other possible file errors. You could also check for each error individually, reporting more specifically the problem that occurred. For the sake of simplicity, the examples in this book will catch only **FileNotFoundException** or **IOException**, but your real-world code may need to handle the other possible exceptions, depending upon the circumstances.

As mentioned, the **FileStream** constructor just described opens a file that has read/ write access. If you want to restrict access to just reading or just writing, use this constructor instead:

```
FileStream(string fi lename, FileMode mode, FileAccess how)
```

As before, *filename* specifies the name of the file to open, and *mode* specifies how the file will be opened. The value passed in *how* determines how the file can be accessed. It must be one of the values defined by the **FileAccess** enumeration, which are shown here:

FileAccess.Read	FileAccess.Write	FileAccess.ReadWrite

For example, this opens a read-only file:

```
FileStream fin = new FileStream("test.dat",
FileMode.Open, FileAccess.Read);
```

When you are done with a file, you should close it by calling **Close( )**. Its general form is shown here:

```
void Close( )
```

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. **Close( )** can throw an **IOException**.

Reading Bytes from a FileStream

FileStream defines two methods that read bytes from a file: **ReadByte( )** and **Read( )**. To read a single byte from a file, use **ReadByte( )**, whose general form is shown here:

```
int ReadByte( )
```

Each time it is called, it reads a single byte from the file and returns it as an integer value. It returns −1 when the end of the file is encountered. Possible exceptions include **NotSupportedException** (the stream is not opened for input) and **ObjectDisposedException** (the stream is closed).

To read a block of bytes, use **Read( )**, which has this general form:

```
int Read(byte[ ] buf, int offset, int numBytes)
```

Read( ) attempts to read up to *numBytes* bytes into *buf* starting at *buf*[*offset*]. It returns the number of bytes successfully read. An **IOException** is thrown if an I/O error occurs. Several other types of exceptions are possible, including **NotSupportedException**, which is thrown if reading is not supported by the stream.

The following program uses **ReadByte( )** to input and display the contents of a text file, the name of which is specified as a command-line argument. Note the **try**/**catch** blocks that handle two errors that might occur when this program is first executed: the specified file not being found or the user forgetting to include the name of the file. You can use this same approach any time you use command-line arguments.

```
/* Display a text file.

   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.CS,
   use the following command line.

   ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;

    try {
```

```
      fin = new FileStream(args[0], FileMode.Open);
    } catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUsage: ShowFile
File");
      return;
    }

    // read bytes until EOF is encountered
    do {
      try {
        i = fin.ReadByte();
      } catch(Exception exc) {
        Console.WriteLine(exc.Message);
        return;
      }
      if(i != -1) Console.Write((char) i);
    } while(i != -1);

    fin.Close();
  }
}
```

Writing to a File

To write a byte to a file, use the **WriteByte( )** method. Its simplest form is shown here:

```
void WriteByte(byte val)
```

This method writes the byte specified by *val* to the file. If an error occurs during writing, an **IOException** is thrown. If the underlying stream is not opened for output, a **NotSupportedException** is thrown. If the stream is closed, **ObjectDisposedException** is thrown.

You can write an array of bytes to a file by calling **Write( )**. It is shown here:

```
void Write(byte[ ] buf, int offset, int numBytes)
```

Write( ) writes *numBytes* bytes from the array *buf*, beginning at *buf*[*offset*], to the file. If an error occurs during writing, an **IOException** is thrown. If

the underlying stream is not opened for output, a **NotSupportedException** is thrown. Several other exceptions are also possible.

As you may know, when file output is performed, often that output is not immediately written to the actual physical device. Instead, output is buffered by the operating system until a sizable chunk of data can be written all at once. This improves the efficiency of the system. For example, disk files are organized by sectors, which might be anywhere from 128 bytes long, on up. Output is usually buffered until an entire sector can be written all at once. However, if you want to cause data to be written to the physical device whether the buffer is full or not, you can call **Flush( )**, shown here:

```
void Flush( )
```

An **IOException** is thrown on failure. If the stream is closed, **ObjectDisposedException** is thrown.

Once you are done with an output file, you must remember to close it using **Close( )**. Doing so ensures that any output remaining in a disk buffer is actually written to the disk. Thus, there is no reason to call **Flush( )** before closing a file.

Here is a simple example that writes to a file:

```
// Write to a file.

using System;
using System.IO;
class WriteToFile {
  public static void Main(string[] args) {
    FileStream fout;

    // open output file
    try {
      fout = new FileStream("test.txt", FileMode.Create);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nError Opening
Output File");
      return;
    }

    // Write the alphabet to the file.
    try {
```

```
      for(char c = 'A'; c <= 'Z'; c++)
        fout.WriteByte((byte) c);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message + "File Error");
    }

    fout.Close();
  }
}
```

The program first opens a file called **test.txt** for output. It then writes the uppercase alphabet to the file. Finally, it closes the file. Notice how possible errors are handled by the **try**/**catch** blocks. After this program executes, **test.txt** will contain the following output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Using FileStream to Copy a File

One advantage to the byte-oriented I/O used by **FileStream** is that you can use it on any type of file—not just those that contain text. For example, the following program copies any type of file, including executable files. The names of the source and destination files are specified on the command line.

```
/* Copy a file.

   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.DAT
   to a file called SECOND.DAT, use the following
   command line.

   CopyFile FIRST.DAT SECOND.DAT
*/

using System;
using System.IO;

class CopyFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;
    FileStream fout;

    try {
```

```
      // open input file
      try {
        fin = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message + "\nInput File Not
Found");
        return;
      }

      // open output file
      try {
        fout = new FileStream(args[1], FileMode.Create);
      } catch(IOException exc) {
        Console.WriteLine(exc.Message + "\nError Opening
Output File");
        return;
      }
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUsage: CopyFile
From To");
      return;
    }

    // Copy File
    try {
      do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte)i);
      } while(i != -1);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message + "File Error");
    }

    fin.Close();
    fout.Close();
  }
}
```

Lab 9: Creating Class Libraries "Dynamic Link Library (DLL)"

We have seen that classes from preexisting libraries, such as the .NET Framework Class Library, can be imported into a C# application. Each class in the FCL belongs to a namespace that contains a group of related classes. As applications become more complex, namespaces help you manage the complexity of application components. Class libraries and namespaces also facilitate software reuse by enabling applications to add classes from other namespaces. This lab introduces how to create your own class libraries.

Steps for Declaring and Using a Reusable Class

Before a class can be used in multiple applications, it must be placed in a class library to make it reusable. Figure 6.1 shows how to specify the namespace in which a class should be placed in the library. Figure 6.4 shows how to use our class library in an application. The steps for creating a reusable class are:

1. Declare a `public` class. If the class is not `public`, it can be used only by other classes in the same assembly.

2. Choose a namespace name and add a `namespace` declaration to the source-code file for the reusable class declaration.

3. Compile the class into a class library.

4. Add a reference to the class library in an application.

5. Specify a `using` directive for the namespace of the reusable class and use the class.

Figure 9.1. `Time1` class declaration in a namespace.

```
 1 // Fig. 9.1: Time1.cs
 2 // Time1 class declaration in a namespace.
 3 namespace lab6
 4 {
 5    public class Time1
 6    {
 7       private int hour; // 0 - 23
 8       private int minute; // 0 - 59
 9       private int second; // 0 - 59
10
11       // set a new time value using universal
time; ensure that
12       // the data remains consistent by setting
invalid values to zero
13       public void SetTime( int h, int m, int s )
14       {
15          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
// validate hour
16          minute = ( ( m >= 0 && m < 60 ) ? m : 0
); // validate minute
17          second = ( ( s >= 0 && s < 60 ) ? s : 0
); // validate second
18       } // end method SetTime
19
20       // convert to string in universal-time
format (HH:MM:SS)
21       public string ToUniversalString()
22       {
23          return string.Format(
"{0:D2}:{1:D2}:{2:D2}",
24             hour, minute, second );
25       } // end method ToUniversalString
26
27       // convert to string in standard-time format
(H:MM:SS AM or PM)
28       public override string ToString()
```

```
29       {
30          return string.Format( "{0}:{1:D2}:{2:D2}
{3}",
31             ( ( hour == 0 || hour == 12 ) ? 12 :
hour % 12 ),
32             minute, second, ( hour < 12 ? "AM" :
"PM" ) );
33       } // end method ToString
34    } // end class Time1
35 } // end namespace lab6
```

Step 1: Creating a `public` Class

For Step 1 in this discussion, we use the `public` class `Time1`

Step 2: Adding the `namespace` Declaration

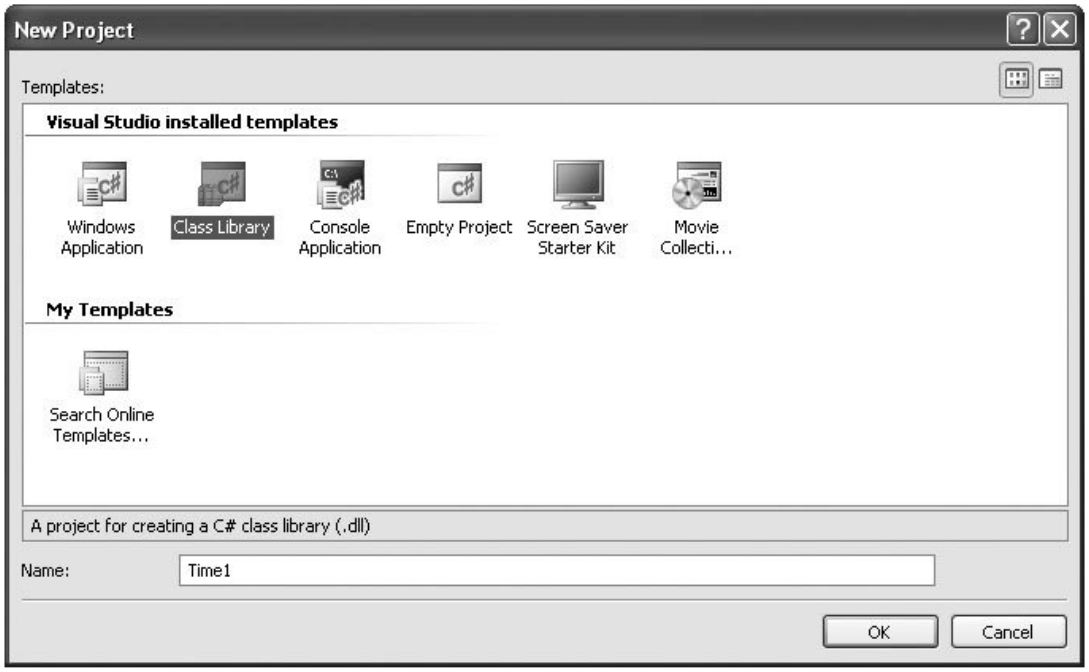
For Step 2, we add a `namespace` declaration to Fig. 10.1.. Line 3 declares a `namespace` named `lab6`. Placing the `Time1` class inside the `namespace` declaration indicates that the class is part of the specified namespace. The `namespace` name is part of the fully qualified class name, so the name of class `Time1` is actually `lab6.Time1`. You can use this fully qualified name in your applications, or you can write a `using` directive (as we will see shortly) and use its simple name (the unqualified class name`Time1`) in the application. If another namespace also contains a `Time1` class, the fully qualified class names can be used to distinguish between the classes in the application and prevent a name conflict (also called a name collision).

Only `namespace` declarations, `using` directives, comments and C# attributes can appear outside the braces of a type declaration (e.g., classes and enumerations). Only class declarations declared `public` will be reusable by clients of the class library. Non-`public` classes are typically placed in a library to support the `public` reusable classes in that library.

Step 3: Compiling the Class Library

Step 3 is to compile the class into a class library. To create a class library in Visual C# Express, we must create a new project by clicking the File menu, selecting New Project... and choosing Class Library from the list of templates, as shown in Fig 10.2. Then add the code from Fig. 10.1 into the new project. In the projects you've created so far, the C# compiler created an executable `.exe` containing the application. When you compile a Class Library project, the compiler creates a `.dll` file, known as a dynamic link library a type of assembly that you can reference from other applications.

Figure 9.2. Creating a Class Library Project.



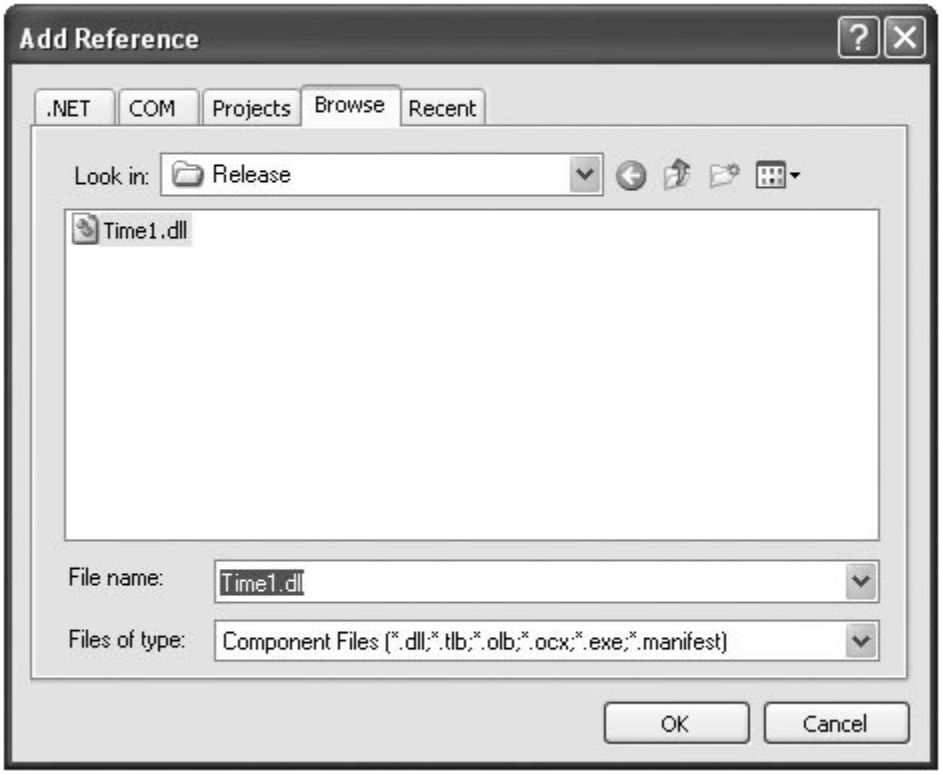
Step 4: Adding a Reference to the Class Library

Once the class is compiled and stored in the class library file, the library can be referenced from any application by indicating to the Visual C# Express IDE where to find the class library file (Step 4). Create a new (empty) project and right-click the project name in the Solution Explorer window.

Select Add Reference... from the pop-up menu that appears. The dialog box that appears will contain a list of class libraries from the .NET Framework. Some class libraries, like the one containing the `System` namespace, are so common that they are added to your application implicitly. The ones in this list are not.

In the Add Reference... dialog box, click the Browse tab. When you build an application, Visual C# 2005 places the `.exe` file in the `bin\Release` folder in the directory of your application. When you build a class library, Visual C# places the `.dll` file in the same place. In the Browse tab, you can navigate to the directory containing the class library file you created in Step 3, as shown in Fig. 10.3. Select the `.dll` file and click OK.

Figure 9.3. Adding a Reference.



Step 5: Using the Class from an Application

Add a new code file to your application and enter the code for class `Time1NamespaceTest` (Fig. 9.4). Now that you've added a reference to your class library in this application, your `Time1` class can be used by `Time1NamespaceTest` (Step 5) without adding the `Time1.cs` source code file to the project.

Figure 9.4. `Time1` object used in an application.

```
 1 // Fig. 9.4: Time1NamespaceTest.cs
 2 // Time1 object used in an application.
 3 using lab10;
 4 using System;
 5
 6 public class Time1NamespaceTest
 7 {
 8    public static void Main( string[] args )
 9    {
10       // create and initialize a Time1 object
11       Time1 time = new Time1(); // calls Time1
constructor
12
13       // output string representations of the time
14       Console.Write( "The initial universal time
is: " );
15       Console.WriteLine( time.ToUniversalString()
);
16       Console.Write( "The initial standard time
is: " );
17       Console.WriteLine( time.ToString() );
18       Console.WriteLine(); // output a blank line
19
20       // change time and output updated time
21       time.SetTime( 13, 27, 6 );
```

```
22       Console.Write( "Universal time after SetTime
is: " );
23       Console.WriteLine( time.ToUniversalString()
);
24       Console.Write( "Standard time after SetTime
is: " );
25       Console.WriteLine( time.ToString() );
26       Console.WriteLine(); // output a blank line
27
28       // set time with invalid values; output
updated time
29       time.SetTime( 99, 99, 99 );
30       Console.WriteLine( "After attempting invalid
settings:" );
31       Console.Write( "Universal time: " );
32       Console.WriteLine( time.ToUniversalString()
);
33       Console.Write( "Standard time: " );
34       Console.WriteLine( time.ToString() );
35    } // end Main
36 } // end class Time1NamespaceTest
```

In Fig.9.4, the `using` directive in line 3 specifies that we'd like to use the class(es) of namespace `lab6` in this file. Class `Time1NamespaceTest` is in the global namespace of this application because the class's file does not contain a `namespace` declaration. Since the two classes are in different namespaces, the `using` directive at line 3 allows class `Time1NamespaceTest` to use class `Time1` as if it was in the same namespace.

We could omit the `using` directive in line 4 if we always referred to class `Console` by its fully qualified class name, `System.Console`. Similarly, we could omit the `using` directive in line 3 for namespace `lab6` if we changed the `Time1` declaration in line 11 of Fig. 9.4 to use class `Time1`'s fully qualified name, as in:

```
Lab9.Time1 time = new lab9.Time1();
```

Lab 10: Multithreading

Objectives

In this lab you will learn:

- What threads are and why they are useful.
- How threads enable you to manage concurrent activities.
- The life cycle of a thread.
- Thread priorities and scheduling.
- To create and execute Threads.
- Thread synchronization.
- What producer/consumer relationships are and how they are implemented with multithreading.

Part 1:

1. Introduction

The .NET Framework Class Library provides concurrency primitives. You specify that applications contain "threads of execution," each of which designates a portion of a program that may execute concurrently with other threads this capability is called multithreading. Multithreading is available to all .NET programming languages, including C#, Visual Basic and Visual C++. The .NET Framework Class Library includes multithreading capabilities in namespace System.Threading.

We discuss many applications of concurrent programming. When programs download large files, such as audio clips or video clips over the Internet, users do not want to wait until an entire clip downloads before starting the playback. To solve this problem, we can put multiple threads to work, one thread downloads a clip, while another plays the clip. These activities proceed concurrently. To avoid choppy playback, we synchronize the threads so that the player thread does not begin until there is a sufficient amount of the clip in memory to keep the player thread busy.

2. Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several thread states that are illustrated in the UML state diagram of Fig. 10.1. This section discusses these states and the transitions between states. Two classes critical for multithreaded applications are `Thread` and `Monitor` (`System.Threading` namespace). This section also discusses several methods of classes `THRead` and `Monitor` that cause state transitions. A few of the terms in the diagram are discussed in later sections.

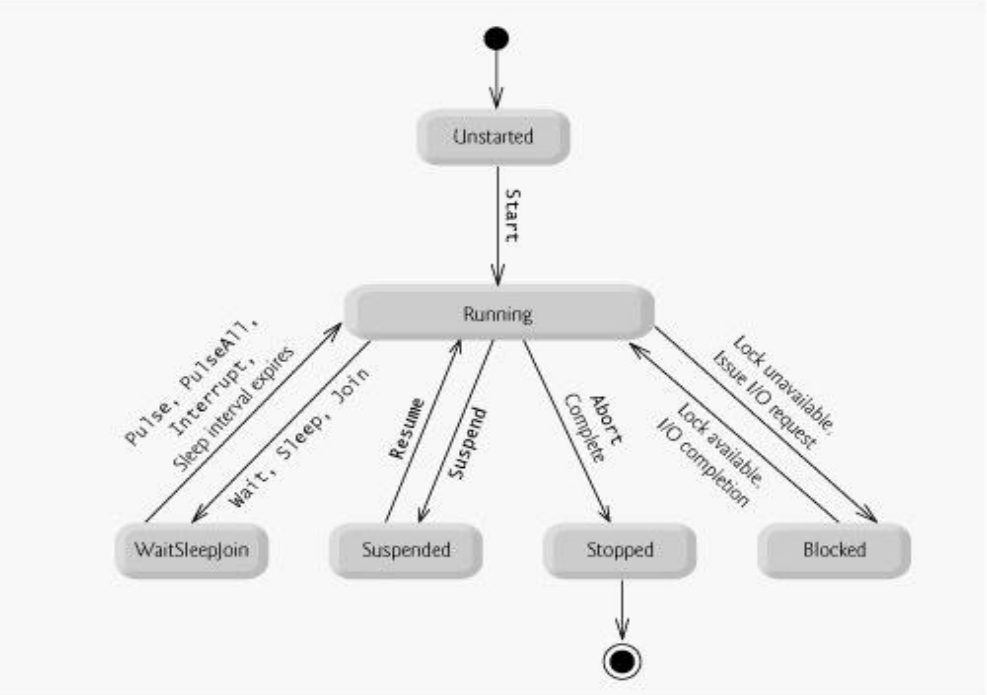


Figure 10.1. Thread life cycle.

A `Thread` object begins its life cycle in the Unstarted state when the program creates the object and passes a ThreadStart delegate to the object's constructor. The `THReadStart` delegate, which specifies the actions the thread will perform during its life cycle, must be initialized with a method that returns `void` and takes no arguments.

[Note: .NET 2.0 also includes a `ParameterizedThreadStart` delegate to which you can pass a method that takes arguments.]

The thread remains in the *Unstarted* state until the program calls the *Thread*'s *Start* method, which places the thread in the Running state and immediately returns control to the part of the program that called Start. Then the newly Running thread and any other threads in the program can execute concurrently on a multiprocessor system or share the processor on a system with a single processor.

While in the *Running* state, the thread may not actually be executing all the time. The thread executes in the Running state only when the operating system assigns a processor to the thread. When a Running thread receives a processor for the first time, the thread begins executing the method specified by its ThreadStart delegate.

A Running thread enters the Stopped (or Aborted) state when its THReadStart delegate terminates, which normally indicates that the thread has completed its task. Note that a program can force a thread into the Stopped state by calling Thread method Abort on the appropriate THRead object. Method Abort throws a ThreadAbortException in the thread, normally causing the thread to terminate. When a thread is in the Stopped state and there are no references to the thread object, the garbage collector can remove the thread object from memory.

[Note: Internally, when a thread's Abort method is called, the thread actually enters the AbortRequested state before entering the Stopped state. The thread remains in the AbortRequested state while waiting to receive the pending ThreadAbortException. When Abort is called, if the thread is in the WaitSleepJoin, *Suspended* or *Blocked* state, the thread resides in its current state and the AbortRequested state, and cannot receive the ThreadAbortException until it leaves its current state.]

A thread is considered Blocked if it is unable to use a processor even if one is available. For example, a thread becomes blocked when it issues an input/output (I/O) request. The operating system blocks the thread from executing until the operating system can complete the I/O request for which the thread is waiting. At that point, the thread returns to the Running state, so it can resume execution. Another case in which a thread becomes blocked is in thread synchronization. A thread being synchronized must acquire a lock on an object by calling Monitor method Enter. If a lock is not available, the thread is blocked until the desired lock becomes available.

[Note: The Blocked state is not an actual state in .NET. It is a conceptual state that describes a thread that is not Running.]

There are three ways in which a Running thread enters the WaitSleepJoin state. If a thread encounters code that it cannot execute yet (normally because a condition is not satisfied), the thread can call *`Monitor`* method *`Wait`* to enter the *WaitSleepJoin* state. Once in this state, a thread returns to the Running state when another thread invokes `Monitor` method `Pulse` or `PulseAll`. Method `Pulse` moves the next waiting thread back to the Running state. Method `PulseAll` moves all waiting threads back to the Running state.

A Running thread can call `Thread` method `Sleep` to enter the WaitSleepJoin state for a period of milliseconds specified as the argument to `Sleep`. A sleeping thread returns to the Running state when its designated sleep time expires. Sleeping threads cannot use a processor, even if one is available.

Any thread that enters the WaitSleepJoin state by calling `Monitor` method `Wait` or by calling `Thread` method `Sleep` also leaves the WaitSleepJoin state and returns to the Running state if the sleeping or waiting *`THRead`*'s *`Interrupt`* method is called by another thread in the program. The `Interrupt` method causes a `THReadInterruptionException` to be thrown in the interrupted thread.

If a thread cannot continue executing (we will call this the dependent thread) unless another thread terminates, the dependent thread calls the other thread's `Join` method to "join" the two threads. When two threads are "joined," the dependent thread leaves the WaitSleepJoin state and re-enters the Running state when the other thread finishes execution (enters the Stopped state).

If a Running `THRead`'s `Suspend` method is called, the Running thread enters the Suspended state. A Suspended thread returns to the Running state when another thread in the program invokes the Suspended thread's `Resume` method.

[Note: Internally, when a thread's `Suspend` method is called, the thread actually enters the SuspendRequested state before entering the Suspended state. The thread remains in the SuspendRequested state while waiting to respond to the `Suspend` request. If the thread is in the WaitSleepJoin state or is blocked when its `Suspend` method is called, the thread resides in its current state and the SuspendRequested state, and cannot respond to the `Suspend` request until it leaves its current state.]

Methods `Suspend` and `Resume` are now deprecated (مهملة؛ متروكة) and should not be used.

If a thread's `IsBackground` property is set to `true`, the thread resides in the Background state (not shown in Fig. 11.1). A thread can reside in the Background state and any other state simultaneously. A process must wait for all foreground threads (threads not in the Background state) to finish executing and enter the Stopped state before the process can terminate. However, if the only threads remaining in a process are Background threads, the CLR terminates each thread by invoking its `Abort` method, and the process terminates.

3. Thread Priorities and Thread Scheduling

Every thread has a priority in the range between *`ThreadPriority.Lowest`* to *`ThreadPriority.Highest`*. These values come from the `ThreadPriority` enumeration (namespace `System.Threading`), which consists of the values `Lowest`, `BelowNormal`, `Normal`, `AboveNormal` and `Highest`. By default, each thread has priority `Normal`.

The Windows operating system supports a concept, called timeslicing, that enables threads of equal priority to share a processor. Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless the thread leaves the Running state and enters the WaitSleepJoin, Suspended or Blocked state) before the thread's peers get a chance to execute. With timeslicing, each thread receives a brief burst of processor time, called a quantum, during which the thread can execute. At the completion of the quantum, even if the thread has not finished executing, the processor is taken away from that thread and given to the next thread of equal priority, if one is available.

The job of the thread scheduler is to keep the highest-priority thread running at all times and, if there is more than one highest-priority thread, to ensure that all such threads execute for a quantum in round-robin fashion. Figure 11.2 illustrates the multilevel priority queue for threads. In Fig. 11.2, assuming a single-processor computer, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution. This means that A gets a quantum of time to run. Then B gets a quantum. Then A gets another quantum. Then B gets another quantum. This continues until one thread completes. The processor then devotes all its power to the thread that remains (unless another thread of that priority is started). Next, thread C runs to completion. Threads D, E and F each execute for a quantum in round-robin fashion until they all complete execution. This process continues until all threads run to completion. Note that, depending on the operating system, new higher-priority threads could postpone (يؤجّل) possibly indefinitely (لاجل غير مسمى) the execution of lower-priority threads. Such indefinite postponement (تأجيل) often is referred to more colorfully as starvation (جوع).

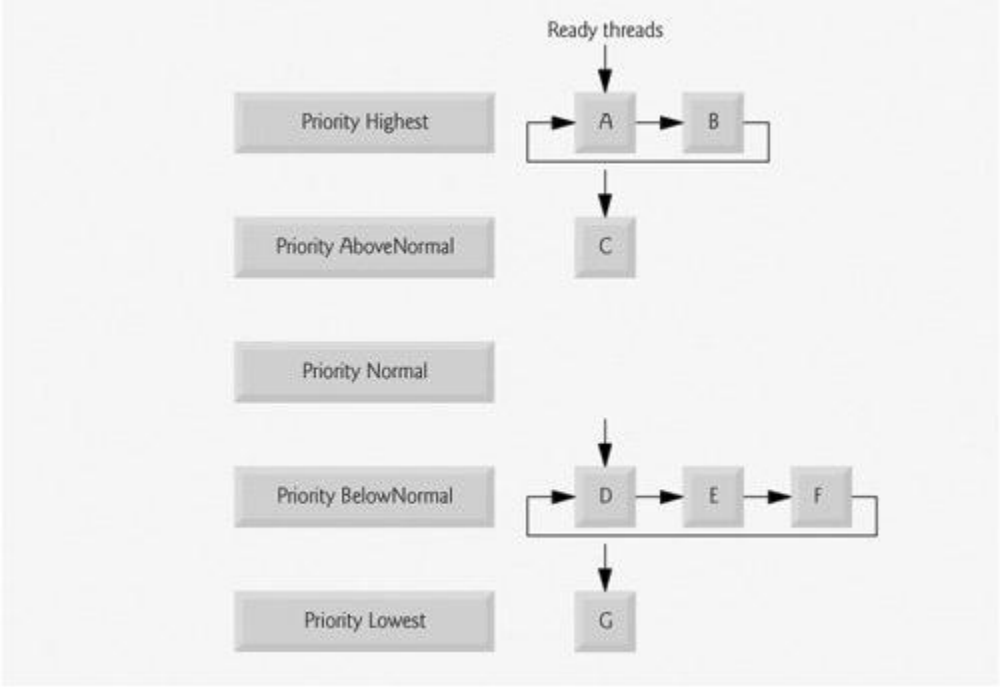


Figure 10.2. Thread-priority scheduling.

A thread's priority can be adjusted with the `Priority` property, which accepts values from the *`ThreadPriority`* enumeration. If the value specified is not one of the valid thread-priority constants, an `ArgumentException` occurs.

A thread executes until it dies, becomes Blocked for I/O (or some other reason), calls `Sleep`, calls `Monitor` method `Wait` or `Join`, is pre-empted (أخذت الاسبقية منه) by a thread of higher priority or has its quantum expire. A thread with a higher priority than the Running thread can become Running (and hence pre-empt (يأخذ ألأسبقية من) the first Running thread) if a sleeping thread wakes up, if I/O completes for a thread that Blocked for that I/O, if either `Pulse` or `PulseAll` is called on an object on which `Wait` was called, if a thread is `Resume`d from the Suspended state or if a thread to which the high-priority thread was joined completes.

4. Creating and Executing Threads

Figure 10.3 demonstrates basic threading techniques, including constructing `THRead` objects and the use of the `Thread` class's `static` method `Sleep`. The program creates three threads of execution, each with the default priority `Normal`. Each thread displays a message indicating that it is going to sleep for a random interval of from 0 to 5000 milliseconds, then goes to sleep. When each thread awakens, the thread displays its name, indicates that it is done sleeping, terminates and enters the Stopped state. You will see that method `Main` (i.e., the `Main` *thread of execution*) terminates before the application terminates. The program consists of two classes `ThreadTester` (lines 7 - 35), which creates the three threads, and `MessagePrinter` (lines 38 - 64), which defines a `Print` method containing the actions each thread will perform.

```
 1 // Fig. 10.3: ThreadTester.cs
 2 // Multiple threads printing at different intervals.
 3 using System;
 4 using System.Threading;
 5
 6 // class ThreadTester demonstrates basic threading concepts
 7 class ThreadTester
 8 {
 9    static void Main( string[] args )
10    {
11       // Create and name each thread. Use MessagePrinter's
12       // Print method as argument to ThreadStart delegate.
```

```
13       MessagePrinter printer1 = new MessagePrinter();
14       Thread thread1 = new Thread ( new ThreadStart(
printer1.Print ) );
15       thread1.Name = "thread1";
16
17       MessagePrinter printer2 = new MessagePrinter();
18       Thread thread2 = new Thread ( new ThreadStart(
printer2.Print ) );
19       thread2.Name = "thread2";
20
21       MessagePrinter printer3 = new MessagePrinter();
22       Thread thread3 = new Thread ( new ThreadStart(
printer3.Print ) );
23       thread3.Name = "thread3";
24
25       Console.WriteLine( "Starting threads" );
26
27       // call each thread's Start method to place each
28       // thread in Running state
29       thread1.Start();
30       thread2.Start();
31       thread3.Start();
32
33       Console.WriteLine( "Threads started\n" );
34    } // end method Main
35 } // end class ThreadTester
36
37 // Print method of this class used to control threads
38 class MessagePrinter
39 {
40    private int sleepTime;
41    private static Random random = new Random();
42
43    // constructor to initialize a MessagePrinter object
44    public MessagePrinter()
45    {
46       // pick random sleep time between 0 and 5 seconds
47       sleepTime = random.Next( 5001 ); // 5001 milliseconds
48    } // end constructor
49
50    // method Print controls thread that prints messages
51    public void Print()
52    {
53       // obtain reference to currently executing thread
54       Thread current = Thread.CurrentThread;
55
56       // put thread to sleep for sleepTime amount of time
57       Console.WriteLine( "{0} going to sleep for {1}
milliseconds",
58          current.Name, sleepTime );
```

```
59          Thread.Sleep( sleepTime ); // sleep for sleepTime
milliseconds
60
61          // print thread name
62          Console.WriteLine( "{0} done sleeping", current.Name );
63       } // end method Print
64 } // end class MessagePrinter
```

Objects of class `MessagePrinter` (lines 38 - 64) control the life cycle of each of the three threads created in class `ThreadTester`'s `Main` method. Class `MessagePrinter` consists of instance variable `sleepTime` (line 40), `static` variable `random` (line 41), a constructor (lines 44 - 48) and a `Print` method (lines 51 - 63). Variable `sleepTime` stores a random integer value chosen when a new `MessagePrinter` object's constructor is called. Each thread controlled by a `MessagePrinter` object sleeps for the amount of time specified by the corresponding `MessagePrinter` object's `sleepTime`.

The `MessagePrinter` constructor (lines 44 - 48) initializes `sleepTime` to a random number of milliseconds from 0 up to, but not including, 5001 (i.e., from 0 to 5000).

Method `Print` begins by obtaining a reference to the currently executing thread (line 54) via class `Thread`'s `static` property `CurrentThread`. The currently executing thread is the one that invokes method `Print`. Next, lines 57 - 58 display a message indicating the name of the currently executing thread and stating that the thread is going to sleep for a certain number of milliseconds. Note that line 58 uses the currently executing thread's `Name` property to obtain the thread's name (set in method `Main` when each thread is created). Line 59 invokes `static Thread` method `Sleep` to place the thread in the WaitSleepJoin state. At this point, the thread loses the processor, and the system allows another thread to execute if one is ready to run. When the thread awakens, it re-enters the Running state and waits to be assigned a processor by the thread scheduler. When the `MessagePrinter` object enters the Running state again, line 62 outputs the thread's name in a message that indicates the thread is done sleeping, and method `Print` terminates.

Class `ThreadTester`'s `Main` method (lines 9 - 34) creates three objects of class `MessagePrinter`, at lines 13, 17 and 21, respectively. Lines 14,

18 and 22 create and initialize three `THRead` objects. Note that each `Thread`'s constructor receives a `ThreadStart` delegate as an argument. A `ThreadStart` delegate represents a method with no arguments and a `void` return type that specifies the actions a thread will perform. Line 14 initializes the `THReadStart` delegate for `tHRead1` with `printer1`'s `Print` method. When `tHRead1` enters the Running state for the first time, `tHRead1` will invoke `printer1`'s `Print` method to perform the tasks specified in method `Print`'s body. Thus, `thread1` will print its name, display the amount of time for which it will go to sleep, sleep for that amount of time, wake up and display a message indicating that the thread is done sleeping. At that point, method `Print` will terminate. A thread completes its task when the method specified by its `THReadStart` delegate terminates, at which point the thread enters the Stopped state. When `tHRead2` and `thread3` enter the Running state for the first time, they invoke the `Print` methods of `printer2` and `printer3`, respectively. `thread2` and `tHRead3` perform the same tasks as `thread1` by executing the `Print` methods of the objects to which `printer2` and `printer3` refer (each of which has its own randomly chosen sleep time). Lines 15, 19 and 23 set each `THRead`'s `Name` property, which we use for output purposes.

Lines 29 - 31 invoke each `Thread`'s `Start` method to place the threads in the Running state. Method `Start` returns immediately from each invocation, then line 33 outputs a message indicating that the threads were started, and the `Main` thread of execution terminates. The program itself does not terminate, however, because there are still non-background threads that are alive (i.e., the threads are Running and have not yet reached the Stopped state). The program will not terminate until its last non-background thread dies. When the system assigns a processor to a thread, the thread enters the Running state and calls the method specified by the thread's `ThreadStart` delegate. In this program, each thread invokes method `Print` of the appropriate `MessagePrinter` object to perform the tasks discussed previously.

Note that the sample outputs for this program show each thread and the thread's sleep time in milliseconds as the thread goes to sleep. The thread with the shortest sleep time normally awakens first, then indicates that it is done sleeping and terminates.

Lab 11: Multithreading

Part 2: Thread Synchronization:

When using multiple threads, you will sometimes need to coordinate the activities of two or more of the threads. The process by which this is achieved is called *synchronization.* The most common reason for using synchronization is when two or more threads need access to a shared resource that can be used by only one thread at a time. For example, when one thread is writing to a file, a second thread must be prevented from doing so at the same time. Another situation in which synchronization is needed is when one thread is waiting for an event that is caused by another thread. In this case, there must be some means by which the first thread is held in a suspended state until the event has occurred. Then the waiting thread must resume execution.

Key to synchronization is the concept of a *lock,* which controls access to a block of code within an object. When an object is locked by one thread, no other thread can gain access to the locked block of code. When the thread releases the lock, the code block is available for use by another thread.

The lock feature is built into the C# language. Thus, all objects can be synchronized. Synchronization is supported by the keyword **lock**. Since synchronization was designed into C# from the start, it is much easier to use than you might expect. In fact, for many programs, the synchronization of objects is almost transparent.

The general form of **lock** is shown here:

```
lock(object)  {
    / / statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. If you want to synchronize only a single statement, the curly braces are not needed. A **lock** statement ensures that the section of code protected by the lock for the given object can be used only by the thread that obtains the lock. All other threads are blocked until the lock is removed. The lock is released when the block is exited.

The following program demonstrates synchronization by controlling access to a method called **sumIt( )**, which sums the elements of an integer array:

```
// Use lock to synchronize access to an object.

using System;
using System.Threading;

class SumArray {
  int sum;

  public int sumIt(int[] nums) {
    lock(this) { // lock the entire method
      sum = 0; // reset sum

      for(int i=0; i < nums.Length; i++) {
        sum += nums[i];
        Console.WriteLine("Running total for " +
              Thread.CurrentThread.Name +
              " is " + sum);
        Thread.Sleep(10); // allow task-switch
      }
      return sum;
    }
  }
}

class MyThread {
  public Thread thrd;
  int[] a;
  int answer;

  /* Create one SumArray object for all
     instances of MyThread. */
  static SumArray sa = new SumArray();

  // Construct a new thread.
  public MyThread(string name, int[] nums) {
    a = nums;
    thrd = new Thread(this.run);
    thrd.Name = name;
    thrd.Start(); // start the thread
  }

  // Begin execution of new thread.
  void run() {
```

```
    Console.WriteLine(thrd.Name + " starting.");

    answer = sa.sumIt(a);

    Console.WriteLine("Sum for " + thrd.Name +
                      " is " + answer);

    Console.WriteLine(thrd.Name + " terminating.");
  }
}

class Sync {
  public static void Main() {
    int[] a = {1, 2, 3, 4, 5};

    MyThread mt1 = new MyThread("Child #1", a);
    MyThread mt2 = new MyThread("Child #2", a);

    mt1.thrd.Join();
    mt2.thrd.Join();
  }
}
```

Let's examine this program in detail. The program creates three classes. The first is **SumArray**. It defines the method **sumIt( )**, which sums an integer array. The second class is **MyThread**, which uses a **static** object called **sa** that is of type **SumArray**. Thus, only one object of **SumArray** is shared by all objects of type **MyThread**. This object is used to obtain the sum of an integer array. Notice that **SumArray** stores the running total in a field called **sum**. Thus, if two threads use **sumIt( )** concurrently, both will be attempting to use **sum** to hold the running total. Since this will cause errors, access to **sumIt( )** must be synchronized. Finally, the class **Sync** creates two threads and has them compute the sum of an integer array.

Inside **sumIt( )**, the **lock** statement prevents simultaneous use of the method by different threads. Notice that **lock** uses **this** as the object being synchronized. This is the way **lock** is normally called when the invoking object is being locked. **Sleep( )** is called to purposely allow a task-switch to occur, if one can—but it can't in this case. Because the code within **sumIt( )** is locked, it can be used by only one thread at a time. Thus, when the second child thread begins execution, it does not enter **sumIt( )** until after the first child thread is done with it. This ensures that the correct result is produced.

To understand the effects of **lock** fully, try removing it from the body of **sumIt( )**. After doing this, **sumIt( )** is no longer synchronized, and any number of threads can use it concurrently on the same object. The problem with this is that the running total is stored in **sum**, which will be changed by each thread that calls **sumIt( )**. Thus, when two threads call **sumIt( )** at the same time on the same object, incorrect results are produced because **sum** reflects the summation of both threads, mixed together.

The effects of **lock** are summarized here:

1. For any given object, once a lock has been placed on a section of code, the object is locked and no other thread can acquire the lock.
2. Other threads trying to acquire the lock on the same object will enter a wait state until the code is unlocked.
3. When a thread leaves the locked block, the object is unlocked.

One other thing to understand about **lock** is that it should not be used on public types or public object instances. Otherwise, some thread external to your program could alter the state of the lock in ways not understood by your program. Thus, a construct such as **lock(this)** should not be used on systemwide objects.

Lab 12: Multithreading

Part 3: Thread Communication Using Wait( ), Pulse( ), and PulseAll( )

Consider the following situation. A thread called *T* is executing inside a **lock** block and needs access to a resource, called *R*, that is temporarily unavailable. What should *T* do? If *T* enters some form of polling loop that waits for *R*, then *T* ties up the object, blocking other threads' access to it. This is a less than optimal solution because it partially defeats the advantages of programming for a multithreaded environment. A better solution is to have *T* temporarily relinquish control of the object, allowing another thread to run. When *R* becomes available, *T* can be notified and resume execution. Such an approach relies upon some form of interthread communication in which one thread can notify another that it is blocked, and be notified when it can resume execution. C# supports interthread communication with the **Wait( )**, **Pulse( )**, and **PulseAll( )** methods.

The **Wait( )**, **Pulse( )**, and **PulseAll( )** methods are defined by the **Monitor** class. These methods can be called only from within a locked block of code. Here is how they are used. When a thread is temporarily blocked from running, it calls **Wait( )**. This causes the thread to go to sleep and the lock for that object to be released, allowing another thread to use the object. At a later point, the sleeping thread is awakened when some other thread calls **Pulse( )** or **PulseAll( )**. A call to **Pulse( )** resumes the first thread in the queue of threads waiting for the lock. A call to **PulseAll( )** signals the release of the lock to all waiting threads.

Here are two commonly used forms of **Wait( )**:

```
public static bool Wait(object waitOb)
public static bool Wait(object waitOb, int milliseconds)
```

The first form waits until notified. The second form waits until notified or until the specified period of milliseconds has expired. For both, *waitOb* specifies the object upon which to wait.

Here are the general forms for **Pulse( )** and **PulseAll( )**:

```
public static void Pulse(object waitOb)
public static void PulseAll(object waitOb)
```

Here, *waitOb* is the object being released.

A **SynchronizationLockException** will be thrown if **Wait( )**, **Pulse( )**, or **PulseAll( )** is called from code that is not within a **lock** block.

An Example That Uses Wait( ) and Pulse( )

To understand the need for and the application of **Wait( )** and **Pulse( )**, we will create a program that simulates the ticking of a clock by displaying the words "Tick" and "Tock" on the screen. To accomplish this, we will create a class called **TickTock** that contains two methods: **tick( )** and **tock( )**. The **tick( )** method displays the word "Tick" and **tock( )** displays "Tock". To run the clock, two threads are created, one that calls **tick( )** and one that calls **tock( )**. The goal is to make the two threads execute in a way that the output from the program displays a consistent "Tick Tock"—that is, a repeated pattern of one "Tick" followed by one "Tock".

```
// Use Wait() and Pulse() to create a ticking clock.

using System;
using System.Threading;
class TickTock {

  public void tick(bool running) {
    lock(this) {
      if(!running) { // stop the clock
        Monitor.Pulse(this); // notify any waiting threads
        return;
      }

      Console.Write("Tick ");
      Monitor.Pulse(this); // let tock() run

      Monitor.Wait(this); // wait for tock() to complete
    }
  }

  public void tock(bool running) {
    lock(this) {
      if(!running) { // stop the clock
        Monitor.Pulse(this); // notify any waiting threads
```

```
        return;
      }

      Console.WriteLine("Tock");
      Monitor.Pulse(this); // let tick() run

      Monitor.Wait(this); // wait for tick() to complete
    }
  }
}

class MyThread {
  public Thread thrd;
  TickTock ttOb;

  // Construct a new thread.
  public MyThread(string name, TickTock tt) {
    thrd = new Thread(this.run);
    ttOb = tt;
    thrd.Name = name;
    thrd.Start();
  }

  // Begin execution of new thread.
  void run() {
    if(thrd.Name == "Tick") {
      for(int i=0; i<5; i++) ttOb.tick(true);
      ttOb.tick(false);
    }
    else {
      for(int i=0; i<5; i++) ttOb.tock(true);
      ttOb.tock(false);
    }
  }
}
class TickingClock {
  public static void Main() {
    TickTock tt = new TickTock();
    MyThread mt1 = new MyThread("Tick", tt);
    MyThread mt2 = new MyThread("Tock", tt);

    mt1.thrd.Join();
    mt2.thrd.Join();
    Console.WriteLine("Clock Stopped");
  }
}
```

Let's take a close look at this program. In **Main( )**, a **TickTock** object called **tt** is created, and this object is used to start two threads of execution. Inside the **run( )** method of **MyThread**, if the name of the thread is "Tick", calls to **tick( )** are made. If the name of the thread is "Tock", the **tock( )** method is called. Five calls that pass **true** as an argument are made to each method. The clock runs as long as **true** is passed. A final call that passes **false** to each method stops the clock.

The most important part of the program is found in the **tick( )** and **tock( )** methods. We will begin with the **tick( )** method, which, for convenience, is shown here:

```
public void tick(bool running) {
  lock(this) {
    if(!running) { // stop the clock
      Monitor.Pulse(this); // notify any waiting threads
      return;
    }

    Console.Write("Tick ");
    Monitor.Pulse(this); // let tock() run

    Monitor.Wait(this); // wait for tock() to complete
  }
}
```

First, notice that the code in **tick( )** is contained within a **lock** block. Recall, **Wait( )** and **Pulse( )** can be used only inside synchronized blocks. The method begins by checking the value of the **running** parameter. This parameter is used to provide a clean shutdown of the clock. If it is **false**, then the clock has been stopped. If this is the case, a call to **Pulse( )** is made to enable any waiting thread to run. We will return to this point in a moment. Assuming that the clock is running when **tick( )** executes, the word "Tick" is displayed, and then a call to **Pulse( )** takes place followed by a call to **Wait( )**. The call to **Pulse( )** allows a thread waiting on the same object to run. The call to **Wait( )** causes **tick( )** to suspend until another thread calls **Pulse( )**. Thus, when **tick( )** is called, it displays one "Tick", lets another thread run, and then suspends.

The **tock( )** method is an exact copy of **tick( )**, except that it displays "Tock". Thus, when entered, it displays "Tock", calls **Pulse( )**, and then waits. When viewed as a pair, a call to **tick( )** can be followed only by a call to **tock( )**, which can be followed only by a call to **tick( )**, and so on. Therefore, the two methods are mutually synchronized.

The reason for the call to **Pulse( )** when the clock is stopped is to allow a final call to **Wait( )** to succeed. Remember, both **tick( )** and **tock( )** execute a call to **Wait( )** after displaying their message. The problem is that when the clock is stopped, one of the methods will still be waiting. Thus, a final call to **Pulse( )** is required in order for the waiting method to run. As an experiment, try removing this call to **Pulse( )** and watch what happens. As you will see, the program will "hang" and you will need to press CTRL-C to exit. The reason for this is that when the final call to **tock( )** calls **Wait( )**, there is no corresponding call to **Pulse( )** that lets **tock( )** conclude. Thus, **tock( )** just sits there, waiting forever.

Before moving on, if you have any doubt that the calls to **Wait( )** and **Pulse( )** are actually needed to make the "clock" run right, substitute this version of **TickTock** into the preceding program. It has all calls to **Wait( )** and **Pulse( )** removed.

```
// A non-functional version of TickTock.
class TickTock {

  public void tick(bool running) {
    lock(this) {
      if(!running) { // stop the clock
        return;
      }

      Console.Write("Tick ");
    }
  }

  public void tock(bool running) {
    lock(this) {
      if(!running) { // stop the clock
        return;
      }

      Console.WriteLine("Tock");
    }
```

```
  }
}
```

Lab 13: Socket Programming

IP Addresses in C#

.NET defines two classes in the System.Net namespace to handle various types of IP address information:

- IPAddress
- IPEndPoint

IPAddress

An IPAddress object is used to represent a single IP address. This value can then be used in the various socket methods to represent the IP address. The default constructor for IPAddress is as follows:

```
public IPAddress(long address)
```

The default constructor takes a long value and converts it to an IPAddress value. In practice, this default is almost never used. (How many times do you happen to have the long value of an IP address handy?) Instead, several methods in the IPAddress class can be used to create and manipulate IP addresses. Table 13.1 defines these methods.

Table 13.1: IPAddress Methods

Method	**Description**
Equals	Compares two IP addresses
GetHashCode	Returns a hash value for an IPAddress object
GetType	Returns the type of the IP address instance
HostToNetworkOrder	Converts an IP address from host byte order to network byte order
IsLoopBack	Indicates whether the IP address is considered the loopback address
NetworkToHostOrder	Converts an IP address from network byte order to host byte order
Parse	Converts a string to an IPAddress instance
ToString	Converts an IPAddress to a string representation of the dotted decimal format of the IP address

The Parse() method is most often used to create IPAddress instances:

IPAddress *newaddress* = IPAddress.Parse("192.168.1.1");

This format allows you to use a standard dotted quad IP address in string format and convert it to an IPAddress object.

The IPAddress class also provides four read-only fields that represent special IP addresses for use in programs:

Any Used to represent any IP address available on the local system

Broadcast Used to represent the IP broadcast address for the local network

Loopback Used to represent the loopback address of the system

None Used to represent no network interface on the system

Listing 13.1 shows an example program that demonstrates using the IPAddress class methods and fields.

Listing 13.1: The AddressSample.cs program

```
using System;

using System.Net;

class AddressSample

{

  public static void Main ()

  {

    IPAddress test1 = IPAddress.Parse("192.168.1.1");

    IPAddress test2 = IPAddress.Loopback;

    IPAddress test3 = IPAddress.Broadcast;

    IPAddress test4 = IPAddress.Any;

    IPAddress test5 = IPAddress.None;
```

```
  IPHostEntry ihe =
      Dns.GetHostByName(Dns.GetHostName());
  IPAddress myself = ihe.AddressList[0];
  if (IPAddress.IsLoopback(test2))
    Console.WriteLine("The Loopback address is: {0}",  test2.ToString());
  else
  Console.WriteLine("Error obtaining the loopback address");
  Console.WriteLine("The Local IP address is: {0}\n",  myself.ToString());
  if (myself == test2)
    Console.WriteLine("The loopback address is the  same as local address.\n");
  else
    Console.WriteLine("The loopback address is not the local address.\n");
  Console.WriteLine("The test address is: {0}", test1.ToString());
  Console.WriteLine("Broadcast address: {0}",  test3.ToString());
  Console.WriteLine("The ANY address is: {0}",  test4.ToString());
  Console.WriteLine("The NONE address is: {0}",   test5.ToString());
 }
}
```

The AddressSample.cs program shows a few of the things that can be done with IPAddress objects. One of the more interesting ones is the method used to obtain the local IP address:

```
  IPHostEntry ihe =
      Dns.GetHostByName(Dns.GetHostName());
  IPAddress myself = ihe.AddressList[0];
```

This uses the GetHostByName() and GetHostName() methods of the System.Net.Dns class to determine the local IP address(es) and create an IPHostEntry object. For now it is sufficient to say that it contains the AddressList property, which is an array of IPAddress objects. The AddressSample.cs program takes the first address in the list and assigns it to the *myself* IP address object.

What's interesting about this output is the resulting values of the Any and None addresses. These values might look backward to what you would expect: the Any IPAddress object points to the 0.0.0.0 address, which you might think represents nothing. However, this address is most often used when a system has multiple network interfaces and you do not want to bind a socket to any particular interface. The None IPAddress object points to the 255.255.255.255 address, which is often used when a system wants to create a dummy socket and not bind it to any interfaces.

IPEndPoint

The .NET Framework uses the IPEndPoint object to represent a specific IP address/port combination. An IPEndPoint object is used when binding sockets to local addresses, or when connecting sockets to remote addresses. We'll first examine all the pieces of IPEndPoint and then look at a program that puts it to work.

Two constructors are used to create IPEndPoint instances:

- IPEndPoint(long *address*, int *port*)
- IPEndPoint(IPAddress *address*, int *port*)

Both constructors use two parameters: an IP address value, represented as either a long value or an IPAddress object; and the integer port number. As you can probably guess, the most common constructor used is the IPAddress form.

Table 13.2 describes the methods that can be used with IPEndPoint objects.

Table 13.2: IPEndPoint Methods	
Method	**Description**
Create	Creates an EndPoint object from a SocketAddress object
Equals	Compares two IPEndPoint objects
GetHashCode	Returns a hash value for an IPEndPoint object
GetType	Returns the type of the IPEndPoint instance
Serialize	Creates a SocketAddress instance of the IPEndPoint instance
ToString	Creates a string representation of the IPEndPoint instance

The SocketAddress class is a special class within the System.Net namespace. It represents a serialized version of an IPEndPoint object. This class can be used to store an IPEndPoint instance, which can then be re-created using the IPEndPoint.Create() method. The format of the SocketAddress class is as follows:

- 1 byte represents the AddressFamily of the object.
- 1 byte represents the size of the object.
- 2 bytes represent the port number of the object.
- The remaining bytes represent the IP address of the object.

In addition to the methods, the IPEndPoint class also contains three properties that can be set or obtained from an instance:

Address Gets or sets the IP address property

AddressFamily Gets the IP address family

Port Gets or sets the TCP or UDP port number

Each of these properties can be used with an IPEndPoint instance to obtain information about individual parts of the IPEndPoint object. The Address and Port properties can also be used to set the individual values within an existing IPEndPoint object.

There are also two fields that can be used with the IPEndPoint object to obtain the available port ranges from a system:

MaxPort The maximum value that can be assigned to a port number

MinPort The minimum value that can be assigned to a port number

Example of IPEndPoint at Work

Listing 13.2 shows a sample program that demonstrates the IPEndPoint class and its methods, properties, and fields.

Listing 13.2: The IPEndPointSample.cs program

```
using System;

using System.Net;

class IPEndPointSample

{

  public static void Main ()

  {

   IPAddress test1 = IPAddress.Parse("192.168.1.1");

   IPEndPoint ie = new IPEndPoint(test1, 8000);

   Console.WriteLine("The IPEndPoint is: {0}", ie.ToString());

   Console.WriteLine("The AddressFamily is: {0}", ie.AddressFamily);

   Console.WriteLine("The address is: {0}, and the   port is: {1}\n", ie.Address, ie.Port);

   Console.WriteLine("The min port number is: {0}",   IPEndPoint.MinPort);

   Console.WriteLine("The max port number is: {0}\n",   IPEndPoint.MaxPort);

   ie.Port = 80;

   Console.WriteLine("The changed IPEndPoint value   is: {0}", ie.ToString());

   SocketAddress sa = ie.Serialize();

   Console.WriteLine("The SocketAddress is: {0}",  sa.ToString());
```

```
  }

}
```

The IPEndPointSample.cs program demonstrates several important IPEndPoint features. Note that you can display the complete IPEndPoint object as one string, or you can extract individual parts of the object:

```
Console.WriteLine("The IPEndPoint is: {0}", ie.ToString());

Console.WriteLine("The AddressFamily is: {0}", ie.AddressFamily);

Console.WriteLine("The address is: {0}, and the port is: {1}\n", ie.Address, ie.Port);
```

The program also demonstrates how to change the port value of the IPEndPoint object individually, using the Port property:

ie.Port = 80;

This allows you to change individual address and port values within the object without having to create a new object.

Using C# Sockets

The System.Net.Sockets namespace contains the classes that provide the actual .NET interface to the low-level Winsock APIs. This section gives a brief overview of the C# Socket class.

Subsequent chapters will build on this overview, presenting detailed descriptions and examples of several types of socket programs.

Socket Construction

The core of the System.Net.Sockets namespace is the Socket class. It provides the C# managed code implementation of the Winsock API. The Socket class constructor is as follows:

```
Socket(AddressFamily af, SocketType st,

    ProtocolType pt)
```

As you can see, the basic format of the Socket constructor mimics the original Unix socket() function. It uses three parameters to define the type of socket to create:

- An *AddressFamily* to define the network type
- A *SocketType* to define the type of data connection
- A *ProtocolType* to define a specific network protocol

Each of these parameters is represented by a separate enumeration within the System.Net_.Sockets namespace. Each enumeration contains the values that can be used. For normal IP communications on networks, the AddressFamily.InterNetwork value should always be used for the AddressFamily. With the *InterNetwork AddressFamily*, the *SocketType* parameter must match a particular *ProtocolType* parameter. You are not allowed to mix and match *SocketTypes* and *ProtocolTypes*. Table 13.3 shows the combinations that can be used for IP communications.

Table 13.3: IP Socket Definition Combinations

SocketType	**Protocoltype**	**Description**
Dgram	Udp	Connectionless communication
Stream	Tcp	Connection-oriented communication
Raw	Icmp	Internet Control Message Protocol
Raw	Raw	Plain IP packet communication

Using the enumeration values makes it easy to remember all the options (though it does make for some fairly long Socket() statements!). For example:

```
Socket newsock = Socket(AddressFamily.InterNetwork,

   SocketType.Stream, ProtocolType.Tcp);
```

Several properties of the Socket class can be used to retrieve information from a created Socket object, as described in Table 14.4.

Table 13.4: Socket Properties	
Property	**Description**
AddressFamily	Gets the address family of the Socket
Available	Gets the amount of data that is ready to be read
Blocking	Gets or sets whether the Socket is in blocking mode
Connected	Gets a value that indicates if the Socket is connected to a remote device
Handle	Gets the operating system handle for the Socket
LocalEndPoint	Gets the local EndPoint object for the Socket
ProtocolType	Gets the protocol type of the Socket
RemoteEndPoint	Gets the remote EndPoint information for the Socket
SocketType	Gets the type of the Socket

Note All of the Socket class properties except the LocalEndPoint and RemoteEndPoint are available for a socket immediately after it is created. The LocalEndPoint and RemoteEndPoint properties can only be used on bound sockets.

Listing 13.3 is a simple program that demonstrates the Socket properties. Because the LocalEndPoint property needs a bound Socket object, I used the Bind() method to bind the socket to the loopback address of the system (127.0.0.1).

Listing 13.3: The SockProp.cs sample socket properties program

```
using System;

using System.Net;

using System.Net.Sockets;

class SockProp

{

  public static void Main ()
```

```
  {

   IPAddress ia = IPAddress.Parse("127.0.0.1");

   IPEndPoint ie = new IPEndPoint(ia, 8000);

   Socket test = new Socket(AddressFamily.InterNetwork,  SocketType.Stream,
ProtocolType.Tcp);

   Console.WriteLine("AddressFamily: {0}",  test.AddressFamily);

   Console.WriteLine("SocketType: {0}",   test.SocketType);

   Console.WriteLine("ProtocolType: {0}",  test.ProtocolType);

   Console.WriteLine("Blocking: {0}", test.Blocking);

   test.Blocking = false;

   Console.WriteLine("new Blocking: {0}",test.Blocking);

   Console.WriteLine("Connected: {0}", test.Connected);

   test.Bind(ie);

   IPEndPoint iep = (IPEndPoint)test.LocalEndPoint;

   Console.WriteLine("Local EndPoint: {0}",  iep.ToString());

   test.Close();

  }

}
```

By setting the Blocking property to false, you can use non-blocking sockets.

Using Connection-Oriented Sockets

Once again, the .NET Framework concepts are similar to Unix network programming. In the .NET Framework, you can create connection-oriented communications with remote hosts across a network. Because C# is an object-oriented language, all the Unix socket functions are implemented as

methods of the Socket class. By referring to the method from the Socket instance, you can perform network operations using the indicated socket.

Note This section describes the C# methods in the Sockets class that are used for connection-oriented communication.

The Server Functions

Once a server socket is created, it must be bound to a local network address on the system. The Bind() method is used to perform this function:

Bind(EndPoint *address*)

The *address* parameter must point to a valid IPEndPoint instance, which includes a local IP address and a port number. After the socket is bound to a local address, you use the Listen() method to wait for incoming connection attempts from clients:

Listen(int *backlog*)

The *backlog* parameter defines the number of connections that the system will queue, waiting for your program to service. Any attempts by clients beyond that number of waiting connections will be refused. You should remember that specifying a large number here might have performance consequences for your server. Each pending connection attempt uses buffer space in the TCP buffer area. This means less buffer space available for sent and received packets.

After the Listen() method is performed, the server is ready to accept any incoming connections. This is done with the Accept() method. The Accept() method returns a new socket descriptor, which is then used for all communication calls for the connection.

Here's a sample of C# server code that sets up the necessary socket pieces:

```
IPHostEntry local = Dns.GetHostByName(Dns.GetHostName());

IPEndPoint iep = new IPEndPoint(local.AddressList[0], 8000);

Socket newserver = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
```

```
newserver.Bind(iep);

newserver.Listen(5);

Socket newclient = newserver.Accept();
```

This program will block at the Accept() statement, waiting for a client connection. Once a client connects to the server, the *newclient* Socket object will contain the new connection information and should be used for all communication with the remote client. The *newserver* Socket object will still be bound to the original IPEndPoint object and can be used to accept more connections with another Accept() method. If no more Accept() methods are called, the server will not respond to any more client connection attempts.

After the client connection has been accepted, the client and server can begin transferring data. The Receive() and Send() methods are used to perform this function. Both of these methods are overloaded with four forms of the method. Table 14.5 shows the available methods to use for each.

Table 13.5: The Receive() and Send() Methods

Method	**Description**
Receive(byte[] data)	Receives data and places it in the specified byte array
Receive(byte[] data, SocketFlags sf)	Sets socket attributes, receives data, and places it in the specified byte array
Receive(byte[] data, int size, SocketFlags sf)	Sets socket attributes, receives the specified size of data, and places it in the specified byte array
Receive(byte[] data, int offset, int size, SocketFlags sf)	Sets socket attributes, receives the size bytes of data, and stores it at offset offset in the data byte array
Send(byte[] data)	Sends the data specified in the byte array
Send(byte[] data, SocketFlags sf)	Sets socket attributes and sends the data specified in the bytes array
Send(byte[] data, int size, SocketFlags sf)	Sets socket attributes and sends the specified size of data in the specified byte array
Send(byte[] data, int offset, int size, SocketFlags sf)	Sets socket attributes and sends size bytes of data starting at offset offset in the data byte array

The simple form of the Send() and Receive() methods sends a single byte array of data, or receives data and places it into the specified byte array.

The Client Functions

The client device must also bind an address to the created Socket object, but it uses the Connect() method rather than Bind(). As with Bind(),Connect()requires an IPEndPoint object for the remote device to which the client needs to connect:

```
IPAddress host = IPAddress.Parse("192.168.1.1");

IPEndPoint hostep = new IPEndPoint(host, 8000);

Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

sock.Connect(hostep);
```

The Connect() method will block until the connection has been established. If the connection cannot be established, it will produce an exception (see the section "Socket Exceptions" later in this chapter).

Once the connection has been established, the client can use the Send() and Receive() methods of the Socket classsimilar to the way the server uses them. When communication is done, the Socket instance must be closed. The Socket class usesboth a shutdown() method to gracefully stop a session, and a close() method to actually close the session. The shutdown() method uses one parameter to determine how the socket will shutdown. Available values for Socket.Shutdown() are described in Table 14.6.

Table 13.6: Socket.Shutdown() Values

Value	Description
SocketShutdown.Both	Prevents both sending and receiving data on the socket
SocketShutdown.Receive	Prevents receiving data on the socket. An RST will be sent if additional data is received.
SocketShutdown.Send	Prevents sending data on the socket. A FIN will be sent after all remaining buffer data is sent.

Here is the typical way to gracefully close a connection:

```
sock.Shutdown(SocketShutdown.Both);

sock.Close();
```

This allows the Socket object to gracefully wait until all data has been sent from its internal buffers.

Using Connectionless Sockets

.NET uses the same functionality for connectionless sockets as that employed by the Unix model. When you create a socket with the SocketType.Dgram socket type, the UDP protocol is used to transmit packets across the network. Similar to the Unix model, you must set up the Bind() method for the server to bind the socket to a particular port. Also similar to the Unix model, the server and client do not need to use the Listen() or Connect() methods.

Because there is no connection for communication, the standard Receive() and Send() methods will not work. Instead, you must use the special ReceiveFrom() and SendTo() methods. The formats for these methods comprise the same base parameters as the Receive() and Send() methods (as seen in Table 13.5). In addition, there is an extra parameter that is a reference to an EndPoint object. This parameter defines where the data is going (for SendTo) or where it is coming from (for ReceiveFrom). For example, the simplest format of the methods would be as follows:

```
ReceiveFrom(byte[], ref EndPoint)

SendTo(byte[], ref EndPoint)
```

For UDP communications, the EndPoint object will point to an IPEndPoint object. If you are new to C#, you may not have seen the ref keyword before. The ref keyword indicates that the method will access the EndPoint object by reference in memory, and not by its value. This is a popular technique in C and C^{++} programming, but it is not seen very often in C# programs.

Non-blocking Programming

The .NET Socket class I/O methods use blocking by default. When a program reaches a network function that blocks, such as Receive(), the program waits there until the function completes, such as when data is received from the socket. Three C# techniques are available to avoid using blocking network calls: non-blocking sockets, multiplexed sockets, and asynchronous sockets.

Non-blocking Sockets

As mentioned earlier in the "Using C# Sockets" sections, C# Socket objects contain properties that can be queried for their values. However, one of the properties, Blocking, can also be set. You can set the Blocking property of a socket to false, putting the socket into non-blocking mode.

When the socket is in non-blocking mode, it will not wait for an I/O method to complete. Rather, it will check the method; if it can't be completed, the method will fail and the program will go on. For example, with Blocking set to false, the Receive() method will not wait for data to appear on the socket. Instead, the method will return a value of 0, indicating that no data was available on the socket.

Mutiplexed Sockets

The Socket class provides the Select() method. This method is used to multiplex multiple Socket instances to watch for the ones that are ready to be read or written to. In C#, however, the Select() method is used somewhat differently. Here is the format of the Select() method:

```
Select(IList read, IList write, IList error,

    int microseconds)
```

The *read*, *write*, and *error* parameters are IList objects, which are arrays that contain created sockets to monitor. The *microseconds* parameter defines the amount of time (in microseconds) the Select() method will wait for the events to happen.

The following is a small code fragment showing how the Select() method can be used:

```
ArrayList socketList = new ArrayList(5);

SocketList.Add(sock1);

SocketList.Add(sock2);

Socket.Select(socketList, null, null, 1000);

byte[] buffer = new byte[1024];

for (i = 0; i < socketList.Length - 1; i++)

{

  socketList[i].Receive(buffer);

  ConsoleWriteLine(Encoding.ASCII.GetString(buffer));

}
```

Notice that the Select() method will monitor both *sock1* and *sock2* for incoming data. If no data is present on either socket, the Receive() method will not block the program.

Asynchronous Socket Programming

It is no surprise that the .NET Framework uses the asynchronous socket model introduced by the Windows Winsock API. This method allows you to use a separate method when a socket is ready to receive or send data. Instead of using a Receive() method to wait for data from a client, you can use the BeginReceive() method, which will register a delegate to be called when data is available on the socket. Within the delegate method, you must use the EndReceive() method to stop the asynchronous read and retrieve the data from the socket.

Lab 14: Connection-Oriented Sockets

A Simple TCP Server

you have four tasks to perform before a server can transfer data with a client connection:

- Create a socket
- Bind the socket to a local IPEndPoint
- Place the socket in listen mode
- Accept an incoming connection on the socket

Creating the Server

The first step to constructing a TCP server is to create an instance of a Socket object. The other three functions necessary for successful server operations are then accomplished by using methods of the Socket object. The following C# code snippet includes these steps:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
Socket newsock = Socket(AddressFamily.InterNetwork,
       SocketType.Stream, ProtocolType.Tcp);
newsock.Bind(ipep);
newsock.Listen(10);
Socket client = newsock.Accept();
```

The Socket object created by the Accept() method can now be used to transmit data in either direction between the server and the remote client.

All these basic steps are demonstrated in the SimpleTcpSrvr.cs program, Listing 14.1.

Listing 14.1: The SimpleTcpSrvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class SimpleTcpSrvr
{
  public static void Main()
  {
```

```
    int recv;
    byte[] data = new byte[1024];
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
                9050);
    Socket newsock = new
      Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);
    newsock.Bind(ipep);
    newsock.Listen(10);
    Console.WriteLine("Waiting for a client...");
    Socket client = newsock.Accept();
    IPEndPoint clientep =
           (IPEndPoint)client.RemoteEndPoint;
    Console.WriteLine("Connected with {0} at port {1}",
            clientep.Address, clientep.Port);

    string welcome = "Welcome to my test server";
    data = Encoding.ASCII.GetBytes(welcome);
    client.Send(data, data.Length,
             SocketFlags.None);
    while(true)
    {
      data = new byte[1024];
      recv = client.Receive(data);
      if (recv == 0)
       break;

      Console.WriteLine(
          Encoding.ASCII.GetString(data, 0, recv));
      client.Send(data, recv, SocketFlags.None);
    }
    Console.WriteLine("Disconnected from {0}",
             clientep.Address);
    client.Close();
    newsock.Close();
   }
}
```

First, an empty byte array is defined as a data buffer for incoming and outgoing messages. It is important to remember that the Socket Receive() and Send() methods work only with byte arrays, no matter what type of data is being transmitted. All data transmitted through the Socket must somehow be converted into a byte array. Since this example only uses text strings, the Encoding.ASCII methods found in the System.Text namespace are used to convert between strings and byte arrays and vice versa.

Next, an IPEndPoint object is defined for the local server machine:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
```

By using the IPAddress.Any field, the server will accept incoming connection requests on any network interface that may be configured on the system. If you are interested in accepting packets from only one particular interface, you can specify its IP address individually using the following technique:

```
IPEndPoint ipep = new
  IPEndPoint(IPAddress.Parse("192.168.1.6"), 9050);
```

After determining the proper IPEndPoint object to use, the Socket() constructor can be called to create the TCP socket. Then the Bind() and Listen() methods are used to bind the socket to the new IPEndPoint object and listen for incoming connections.

Finally, the Accept() method is used to accept an incoming connection attempt from a client. The Accept() method returns a new Socket object, which must be used in all communications with the client.

After the Accept() method accepts a connection attempt, the IP address information of the requesting client is available from the returned socket using the RemoteEndPoint property:

```
IPEndPoint clientep =
          (IPEndPoint)client.RemoteEndPoint;
```

Because the RemoteEndPoint property specifies any type of EndPoint object, you must specifically typecast it to an IPEndPoint object. Once the IPEndPoint object is created using the remote client's information, you can access it using the Address and Port properties, as demonstrated in the program. This is a handy technique to use for identifying individual clients that connect to the server.

After the socket is established with the client, the client/server combination must agree upon a system to transfer data. If both the server and client use blocking sockets (the default) and try to receive data at the same time, they will be waiting forever in a deadlock mode. Likewise, they will deadlock if they both send data at the same time. Instead, there must be a "ping pong" effect, where one side sends data while the other side waits to receive it.

In this example, the server immediately sends a welcome message to the client, and then waits for messages from the client:

```
string welcome = "Welcome to my test server";
   data = Encoding.ASCII.GetBytes(welcome);
   client.Send(data, data.Length,
             SocketFlags.None);
   while(true)
   {
     data = new byte[1024];
     recv = client.Receive(data);
     if (recv == 0)
      break;

     Console.WriteLine(
         Encoding.ASCII.GetString(data, 0, recv));
     client.Send(data, recv, SocketFlags.None);
   }
```

Clients wanting to communicate with this server must be prepared to accept the welcome banner as soon as the connection is established. After receiving the welcome banner, the client must alternate between sending and receiving data.

Because the socket sends and receives messages in byte format, you will have to convert any outbound string messages to a byte array type, and any incoming byte arrays to string objects. The easiest way to do this is to use the Encoding class in the System.Text namespace. If you are using ASCII characters for your server, the ASCII property can be used with the GetBytes() or GetString() methods to easily convert strings to byte arrays, and byte arrays to strings like this:

```
Byte[] data = Encoding.ASCII.GetBytes("test string");
string word = Encoding.ASCII.GetString(data);
```

As each message is received from a client, it is immediately sent back to the client. This functionality is called an *echo server* and is often used for testing communication channels. The interesting point to note about this code is how the server knows to stop waiting for more data. This code in the SimpleTcpSrvr program checks the value of the *recv* variable for a zero value.

A Note about the Receive() Method

You may have noticed in Listing 9.1 that the first line in the while loop resets the *data* variable to a new byte array. This is a crucial detail that is forgotten by many less-experienced network programmers. As the Receive() method places data in the data buffer, the size of the data buffer is set. If the data buffer is not reset to its original value, the next Receive() call using the buffer will only be able to place as much data as the previous call.

The Receive() method normally returns the amount of data received from the remote client. If no data is received, the Receive() method blocks, waiting for data to appear. You may be wondering how Receive() can possibly return a value of zero. It's simple: when the TCP structure has detected that the remote client has initiated a close session (by sending a TCP FIN packet), the Receive() method is allowed to return a zero value.

This allows you to check the Receive() return value for a zero value. If it is zero, the program must attempt to gracefully close the newly created client socket by using the Close() method along with the optional Shutdown() method. After the client Socket object is closed, no further communication can take place. Remember however, that the *original* main Socket object is still active and can still be used to accept more incoming connections until it is closed as well. If you do not want any additional connections at all, the original Socket object must also be closed.

Testing the Server

You can easily test the server code using the Microsoft Telnet program that comes standard on all Windows platforms. The Telnet version in Windows 98, Me, and NT is a graphical Windows-based system; the Windows 2000 and XP versions are text-based systems that operate in a command prompt window. The Telnet program allows you to connect via TCP to any address and any port.

To start the sample TCP server, open a command prompt window and type **SimpleTcpSrvr**. The server will display the opening greeting and wait for an incoming client:

```
C:\>SimpleTcpSrvr
Waiting for a client...
```

Once the server is running, open another command prompt window (either on the same system or another system on the network) and start the Telnet program. Connect to the address of the server and the 9050 port used:

```
C:\>telnet 127.0.0.1 9050
```

The connection should start, and the server welcome screen should appear.

At this point, the server is waiting for a message from the client. Using the Telnet program, a strange thing happens. If you try to type a message, each individual character is sent to the server and immediately echoed back. If you try to type in a phrase, each character is sent individually and returned by the server individually. This behavior is caused by the way the Telnet application transmits data to the remote server: it automatically sends each typed character to the remote server. It does not wait for a carriage return or line feed character to transmit data.

A Simple TCP Client

Now that you have a working TCP server, you can create a simple TCP client program to interact with it. There are only two steps required to connect a client program to a TCP server:

- Create a socket
- Connect the socket to the remote server address

This section describes in detail these steps for creating a basic TCP client program using C#.

Creating the Client

As it was for the server program, the first step of creating the client program is to create a Socket object. The Socket object is used by the Socket Connect() method to connect the socket to a remote host:

```
IPEndPoint ipep = new
IPEndPoint(Ipaddress.Parse("192.168.1.6"), 9050);
Socket server = new Socket(AddressFamily.InterNetwork,
       SocketType.Stream, ProtocolType.Tcp);
server.Connect(ipep);
```

This example attempts to connect the socket to the server located at address 192.168.1.6. Of course, you can also use hostnames along with the Dns.Resolve() method. You may have noticed that I didn't use any fancy Exception programming for the server program. We will not have the same walk in the park for the client program. One huge challenge with the client's Connect() method is that if the remote server is unavailable, it will create an Exception. This can result in an ugly error message for customers. It is always a good idea to use a try-catch block to catch SocketExceptions when using the Connect() method so you can provide your own user-friendly message for your customers.

Once the remote server TCP program accepts the connection request, the client program is ready to transmit data with the server using the standard Send() and Receive() methods. Listing 9.2 is the SimpleTcpClient.cs program, which demonstrates these principles.

Listing 14.2: The SimpleTcpClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class SimpleTcpClient
{
  public static void Main()
  {
   byte[] data = new byte[1024];
   string input, stringData;
   IPEndPoint ipep = new IPEndPoint(
           IPAddress.Parse("127.0.0.1"), 9050);
   Socket server = new Socket(AddressFamily.InterNetwork,
           SocketType.Stream, ProtocolType.Tcp);
   try
   {
     server.Connect(ipep);
   } catch (SocketException e)
   {
     Console.WriteLine("Unable to connect to server.");
     Console.WriteLine(e.ToString());
     return;
   }
   int recv = server.Receive(data);
   stringData = Encoding.ASCII.GetString(data, 0, recv);
   Console.WriteLine(stringData);
```

```
    while(true)
    {
      input = Console.ReadLine();
      if (input == "exit")
       break;
      server.Send(Encoding.ASCII.GetBytes(input));
      data = new byte[1024];
      recv = server.Receive(data);
      stringData = Encoding.ASCII.GetString(data, 0, recv);
      Console.WriteLine(stringData);
    }
    Console.WriteLine("Disconnecting from server...");
    server.Shutdown(SocketShutdown.Both);
    server.Close();
   }
}
```

Just like the simple server program, this client program has few frills. It simply creates an IPEndPoint for the server (if you want to connect to a remote server, you will have to plug your server's IP address in the *ipep* IPEndPoint value) and attempts to connect to that server:

```
   IPEndPoint ipep = new IPEndPoint(
              IPAddress.Parse("127.0.0.1"), 9050);
      Socket server = new Socket(AddressFamily.InterNetwork,
              SocketType.Stream, ProtocolType.Tcp);
      try
      {
        server.Connect(ipep);
      } catch (SocketException e)
      {
        Console.WriteLine("Unable to connect to server.");
        Console.WriteLine(e.ToString());
        return;
      }
```

The Connect() method is placed in a try-catch block in hopes that it will ease the pain of the customer if the remote server is not available. Once a connection is established, the program waits for the welcome message sent by the server and displays the message on the console.

After that, the client program enters a loop, taking text entered at the console and sending it as a single message to the server. The resulting message received by the server is echoed to the console. When the input text is equal

to the phrase exit, the while loop exits and the client closes the connection. When the server sees the connection closed—that is, when the Receive() method returns 0—the server, too, exits its while loop and closes the connection.

Testing the Client

The first thing to test is the Exception code used for the situation where the server is unavailable. This is an easy thing to do: just don't start the server program and do run the SimpleTcpClient program. This should produce the warning message you created in the Exception code:

```
C:\>SimpleTcpClient
Unable to connect to server.
System.Net.Sockets.SocketException: Unknown error
(0x274d)
  at System.Net.Sockets.Socket.Connect(EndPoint remoteEP)
  at SimpleTcpClient.Main()
C:\>
```

Obviously, if this were a real production-quality client program, you would want to handle this error with more dignity, possibly allowing the customer to try the connection again. But even this terse bit of code is a much nicer solution than presenting the standard exception message generated by .NET when the SocketException occurs.

Now that the error test is out of the way, it is time to try connecting to the server. First, start the SimpleTcpSrvr program on the designated server machine. Once it has indicated it is waiting for clients, start the SimpleTcpClient program either in a separate command prompt window on the same machine, or on another machine on the network. When the client establishes the TCP connection, it should display the greeting banner from the server. At this point, it is ready to accept data from the console, so you can start entering data.

Notice that the entire phrase you enter at the console is sent to the server and displayed as a single message, and it's then returned back to the client as a single message, where it is also displayed. Because the client Send() method sends the data out as a block of bytes, it is received as a block of bytes at the server's Receive() method. Again, if you are running this test from two separate machines, you can watch this with the WinDump or Analyzer

programs to verify that the data is being sent out as blocks instead of as characters.

Although the simple TCP programs are now behaving as expected, there is still a serious flaw lurking within. These test programs, as they are, use small blocks of data in a strictly controlled test environment, so they most likely worked just fine for you. I say "most likely," because the behavior is not guaranteed.

Using C# Streams with TCP

Because handling messages on a TCP connection is often a challenge for programmers, the .NET Framework supplies some extra classes to help out. This section describes the NetworkStream class, which provides a stream interface for sockets, as well as two additional stream classes, StreamReader and StreamWriter, that can be used to send and receive text messages using TCP.

The NetworkStream Class

C# uses streams to help programmers move data in large chunks. The System.Net.Sockets namespace contains the NetworkStream class, which provides a stream interface to sockets.

There are several constructors that can be used to create a NetworkStream object. The easiest (and most popular) way to create the NetworkStream object is to simply use the Socket object:

```
Socket newsock = new Socket(AddressFamily.InterNetwork,
          SocketType.Stream, ProtocolType.Tcp);
NetworkStream ns = new NetworkStream(newsock);
```

This creates a new NetworkStream object that can then be referenced instead of the Socket object. After the NetworkStream object has been created, there are several properties and methods for augmenting the functionality of the Socket object, listed in Table 15.1.

Table 14.1: NetworkStream Class Properties

Property	Description
CanRead	Is true if the NetworkStream supports reading
CanSeek	Is always false for NetworkStreams
CanWrite	Is true if the NetworkStream supports writing
DataAvailable	Is true if there is data available to be read

The property most often used, DataAvailable,quickly checks to see if there is data waiting in the socket buffer to be read.

The NetworkStream class contains a healthy supply of methods for accessing data in the stream. These methods are listed in Table 15.2.

Table 14.2: NetworkStream Class Methods

Method	Description
BeginRead()	Starts an asynchronous NetworkStream read
BeginWrite()	Starts an asynchronous NetworkStream write
Close()	Closes the NetworkStream object
CreateObjRef()	Creates an object used as a proxy for the NetworkStream
EndRead()	Finishes an asynchronous NetworkStream read
EndWrite()	Finishes an asynchronous NetworkStream write
Equals()	Determines if two NetworkStreams are the same
Flush()	Flushes all data from the NetworkStream
GetHashCode()	Obtains a hash code for the

Table 14.2: NetworkStream Class Methods

Method	Description
	NetworkStream
GetLifetimeService()	Retrieves the lifetime service object for the NetworkStream
GetType()	Retrieves the type of the NetworkStream
InitializeLifetimeService()	Obtains a lifetime service object to control the lifetime policy for the NetworkStream
Read()	Reads data from the NetworkStream
ReadByte()	Reads a single byte of data from the NetworkStream
ToString()	Returns a string representation
Write()	Writes data to the NetworkStream
WriteByte()	Writes a single byte of data to the NetworkStream

Use the Read() method to read blocks of data from the NetworkStream. Its format is as follows, where *buffer* is a byte array buffer to hold the read data, *offset* is the buffer location at which to start placing the data, and *size* is the number of bytes to be read.

```
int Read(byte[] buffer, int offset, int size)
```

The Read() method returns an integer value representing the number of bytes actually read from the NetworkStream and placed in the buffer.

Similarly, the Write() method format is as follows, where *buffer* is the buffer from which to get the data to send, *offset* is the buffer location at which to start getting data, and *size* is the number of bytes to send:

```
void Write(byte[] buffer, int offset, int size)
```

Listing 9.3 shows another version of the TCP client program that uses the NetworkStream object for communication with the Socket object.

Listing 14.3: The NetworkStreamTcpClient.cs progam

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class NetworkStreamTcpClient
{
  public static void Main()
  {
   byte[] data = new byte[1024];
   string input, stringData;
   int recv;
   IPEndPoint ipep = new IPEndPoint(
           IPAddress.Parse("127.0.0.1"), 9050);
   Socket server = new Socket(AddressFamily.InterNetwork,
           SocketType.Stream, ProtocolType.Tcp);
   try
   {
     server.Connect(ipep);
   } catch (SocketException e)
   {
     Console.WriteLine("Unable to connect to server.");
     Console.WriteLine(e.ToString());
     return;
   }
   NetworkStream ns = new NetworkStream(server);
   if (ns.CanRead)
   {
     recv = ns.Read(data, 0, data.Length);
     stringData = Encoding.ASCII.GetString(data, 0, recv);
     Console.WriteLine(stringData);
   }
   else
   {
     Console.WriteLine("Error: Can't read from this
socket");
     ns.Close();
     server.Close();
     return;
   }
   while(true)
   {
     input = Console.ReadLine();
     if (input == "exit")
      break;
     if (ns.CanWrite)
```

```
      {
       ns.Write(Encoding.ASCII.GetBytes(input), 0,
input.Length);
       ns.Flush();
      }
      recv = ns.Read(data, 0, data.Length);
      stringData = Encoding.ASCII.GetString(data, 0, recv);
      Console.WriteLine(stringData);
    }
    Console.WriteLine("Disconnecting from server...");
    ns.Close();
    server.Shutdown(SocketShutdown.Both);
    server.Close();
   }
}
```

This program creates a NetworkStream object from the Socket object:

```
NetworkStream ns = new NetworkStream(server);
```

Once the NetworkStream object has been created, the Socket object is never again referenced until it is closed at the end of the program. All communication with the remote server is done through the NetworkStream object:

```
recv = ns.Read(data, 0, data.Length);
ns.Write(Encoding.ASCII.GetBytes(input), 0,
input.Length);
ns.Flush();
```

The Flush() method is used after every Write() method to ensure that the data placed in the NetworkStream will immediately be sent to the remote system rather than waiting in the TCP buffer area for more data before being sent.

Since the NetworkStreamTcpClient program sends and receives the same data packets, you can test it with the original SimpleTcpSrvr program presented at the beginning of this chapter (Listing 9.1). It should behave the same way as the SimpleTcpClient program, sending a message to the server and receiving the echoed message back.

Although the NetworkStream object has some additional functionality over the Socket object, it is still limited in how it sends and receives data from the socket. The same unprotected message boundary problems exist, as when using the plain Socket object to send and receive messages. The next section describes two classes that can help you have more control of the data on the socket.

The StreamReader and StreamWriter Classes

The System.IO namespace contains the StreamReader and StreamWriter classes that control the reading and writing of text messages on a stream. Both of these classes can be deployed with a NetworkStream object to help define markers for TCP messages.

The StreamReader class has lots of constructor formats for many applications. The simplest format to use with NetworkStreams is as follows:

```
public StreamReader(Stream stream)
```

The stream variable can reference any type of Stream object, including a NetworkStream object. You have a generous selection of properties and methods available for use with the StreamReader object after it is created, as described in Table 15.3.

Table 14.3: StreamReader Class Methods

Method	**Description**
Close()	Closes the StreamReader object
CreateObjRef()	Creates an object used as a proxy for the StreamReader
DiscardBufferedData()	Discards the current data in the StreamReader
Equals()	Determines if two StreamReader objects are the same
GetHashCode()	Returns a hash code for the StreamReader object
GetLifetimeService()	Retrieves the lifetime service object for the StreamReader

Table 14.3: StreamReader Class Methods

Method	Description
GetType()	Retrieves the type of the StreamReader object
InitializeLifetimeService()	Creates a lifetime service object for the StreamReader
Peek()	Returns the next available byte of data from the stream without removing it from the stream
Read()	Reads one or more bytes of data from the StreamReader
ReadBlock()	Reads a group of bytes from the StreamReader stream and places it in a specified buffer location
ReadLine()	Reads data from the StreamReader object up to and including the first line feed character
ReadToEnd()	Reads the data up to the end of the stream
ToString()	Creates a string representation of the StreamReader object

Similar to StreamReader, the StreamWriter object can be created from a NetworkStream object:

```
public StreamWriter(Stream stream)
```

StreamWriter, too, has several associated properties and methods. And, as expected, most of the common methods of the StreamReader class are also found in the StreamWriter class. Table 15.4 shows the other important methods used for StreamWriter.

Table 14.4: Unique StreamWriter Class Methods

Method	Description
Flush()	Sends all StreamWriter buffer data to the underlying stream
Write()	Sends one or more bytes of data to the underlying stream
WriteLine()	Sends the specified data plus a line feed character to the underlying stream

Tip The most interesting StreamReader method is the ReadLine() method. It reads characters from the stream until it comes across a line feed character. This feature allows you to use the line feed character as a message marker to separate text messages. This greatly simplifies receiving text messages with TCP. You may notice that the WriteLine() method is the perfect match to the StreamReader's ReadLine() method. This was not an accident. Together, these two methods let you create message-based TCP communications using the line feed character as a message marker. This feature greatly simplifies TCP message programming—as long as you are using text messages.

The programs presented in the next two sections show you how to use the StreamReader and StreamWriter classes to modify the TCP server and client programs presented earlier to use text messages.

Stream Server

Listing 14.4 is the StreamTcpSrvr.cs program, which demonstrates using these principles in a server program.

Listing 14.4: The StreamTcpSrvr.cs program

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;
class StreamTcpSrvr
{
  public static void Main()
  {
```

```
   string data;
   IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
   Socket newsock = new Socket(AddressFamily.InterNetwork,
           SocketType.Stream, ProtocolType.Tcp);
   newsock.Bind(ipep);
   newsock.Listen(10);
   Console.WriteLine("Waiting for a client...");
   Socket client = newsock.Accept();
   IPEndPoint newclient =
(IPEndPoint)client.RemoteEndPoint;
   Console.WriteLine("Connected with {0} at port {1}",
           newclient.Address, newclient.Port);
   NetworkStream ns = new NetworkStream(client);
   StreamReader sr = new StreamReader(ns);
   StreamWriter sw = new StreamWriter(ns);
   string welcome = "Welcome to my test server";
   sw.WriteLine(welcome);
   sw.Flush();
   while(true)
   {
     try
     {
      data = sr.ReadLine();
     } catch (IOException)
     {
      break;
     }

     Console.WriteLine(data);
     sw.WriteLine(data);
     sw.Flush();
   }
   Console.WriteLine("Disconnected from {0}",
newclient.Address);
   sw.Close();
   sr.Close();
   ns.Close();
  }
}
```

The StreamTcpSrvr program uses the StreamWriter WriteLine() method to send text messages terminated with a line feed. As is true for the NetworkStream object, it is best to use the Flush() method after each WriteLine() call to ensure that all of the data is sent from the TCP buffer.

You may have noticed one major difference between this program and the original SimpleTcpSrvr program: the way StreamTcpSrvr knows when the remote connection is disconnected. Because the ReadLine() method works on the stream and not the socket, it cannot return a 0 when the remote connection has disconnected. Instead, if the underlying socket disappears, the ReadLine() method will produce an Exception. It is up to you to catch the Exception produced when the socket has been disconnected:

```
try
{
 data = sr.ReadLine();
} catch (IOException)
{
  break;
}
```

Stream Client

Now that you have a server that accepts messages delimited by a line feed, you need a client that can send messages in that format. Listing 15.5 shows the StreamTcpClient program that demonstrates using the StreamReader and StreamWriter objects in a TCP client application.

Listing 14.5: The StreamTcpClient.cs program

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;
class StreamTcpClient
{
  public static void Main()
  {
   string data;
   string input;
   IPEndPoint ipep = new IPEndPoint(
           IPAddress.Parse("127.0.0.1"), 9050);
   Socket server = new Socket(AddressFamily.InterNetwork,
           SocketType.Stream, ProtocolType.Tcp);
   try
   {
     server.Connect(ipep);
```

```
    } catch (SocketException e)
    {
      Console.WriteLine("Unable to connect to server.");
      Console.WriteLine(e.ToString());
      return;
    }
    NetworkStream ns = new NetworkStream(server);
    StreamReader sr = new StreamReader(ns);
    StreamWriter sw = new StreamWriter(ns);
    data = sr.ReadLine();
    Console.WriteLine(data);
    while(true)
    {
      input = Console.ReadLine();
      if (input == "exit")
       break;
      sw.WriteLine(input);
      sw.Flush();
      data = sr.ReadLine();
      Console.WriteLine(data);
    }
    Console.WriteLine("Disconnecting from server...");
    sr.Close();
    sw.Close();
    ns.Close();
    server.Shutdown(SocketShutdown.Both);
    server.Close();
   }
}
```

This version of the client program performs the same functions as the SimpleTcpClient program, but it sends messages terminated with a line feed. There is one important difference for you to observe. If you try using the StreamTcpClient program with the original

SimpleTcpSrvr program, it won't work. That's because the greeting banner produced by the SimpleTcpSrvr program does not include a line feed at the end of the text. Without the line feed, the ReadLine() method in the client program does not complete, and blocks the program execution, waiting for the line feed to appear. Remember this behavior whenever you use the ReadLine() stream method.

Warning	Another point to remember when using the ReadLine() method is to ensure that the data itself does not contain a line feed character.

This will create a false marker for the ReadLine() method and affect the way the data is read.

Lab 15: Connectionless Sockets

A Simple UDP Application

UDP is a connectionless protocol. Therefore, the programmer must do only two things to make a server application ready to send or receive UDP packets:

- Create a Socket object
- Bind the socket to a local IPEndPoint

After these two actions are taken, the socket can be used to either accept incoming UDP packets on the IPEndPoint, or send outgoing UDP packets to any other device on the network. All of the TCP connection requirements are unnecessary with UDP.

Note For client UDP applications that do not need to receive UDP packets on a specific UDP port, you do not have to bind the socket to a specific IPEndPoint—just create the Socket object and send data!

Because there is no connection between remote hosts, the UDP application cannot use the standard Send() and Receive() Socket methods. Instead, two new methods must be used, SendTo() and ReceiveFrom().

SendTo() The SendTo() method specifies the data to send and the IPEndpoint of the destination machine. There are several versions of this method that can be used, based on your requirements:

SendTo(byte[] *data*, EndPoint *Remote*)

This simple version of the SendTo() method sends a byte array data to the EndPoint specified by the variable *Remote*. A slightly more complex version is as follows:

SendTo(byte[] data, SocketFlags Flags, EndPoint *Remote*)

This version allows you to include a SocketFlags object Flags, which specifies any special UDP socket options to use. Use the following version of SendTo()to specify the number of bytes from the byte array to send:

SendTo(byte[*data*], int Size, SocketFlags Flags, EndPoint *Remote*)

The last version of the SendTo() method, with which you can specify a specific offset within the byte array to start sending data, is as follows:

SendTo(byte[] *data*, int Offset, int Size, SocketFlags Flags, EndPoint *Remote*)

ReceiveFrom() The ReceiveFrom() method has the same formats as the SendTo() method, with one important difference: the way the EndPoint object is declared. The basic ReceiveFrom() method is defined as follows:

ReceiveFrom(byte[] data, ref EndPoint Remote)

As usual, a byte array is defined to accept the received data.

What's really of interest here is the second parameter, ref EndPoint. Instead of passing an EndPoint object, you must pass the *reference* to an EndPoint object. Although using references to variables is common in C and C^{++} programs, the structure seen here is not all that common in C# programs. The reference refers to the memory location where the variable is stored, not the value of the variable. The ReceiveFrom() method will place the EndPoint information from the remote device into the EndPoint object memory area you reference.

Both SendTo() and ReceiveFrom() are included in the simple UDP server and client examples in the following sections. These examples demonstrate the basics of connectionless communication with C# sockets.

The UDP Server

Although UDP applications aren't really servers or clients by strict definition, I will call this next application a UDP server for the sake of simplicity. It creates a Socket object and binds it to a set IPEndPoint object so it can wait for incoming packets:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any,
            9050);
Socket newsock = Socket(AddressFamily.InterNetwork,
     SocketType.Dgram, ProtocolType.Udp);
newsock.Bind(ipep);
```

For connectionless communications, you must specify the Dgram SocketType, along with the Udp ProtocolType. Remember, if your application does not need to receive UDP data on a specific UDP port, you do not have to bind the socket to a specific IPEndPoint. However, if you do need to listen to specific port, such as for servers, you must use the Bind() method.

Listing 15.1, the SimpleUdpSrvr.cs program, demonstrates the basic components of a simple UDP server by binding a dedicated socket on the server for other UDP clients to connect to.

Listing 15.1: The SimpleUdpSrvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class SimpleUdpSrvr
{
  public static void Main()
  {
   int recv;
   byte[] data = new byte[1024];
   IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
   Socket newsock = new Socket(AddressFamily.InterNetwork,
         SocketType.Dgram, ProtocolType.Udp);
   newsock.Bind(ipep);
   Console.WriteLine("Waiting for a client...");
   IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
   EndPoint Remote = (EndPoint)(sender);
   recv = newsock.ReceiveFrom(data, ref Remote);
   Console.WriteLine("Message received from {0}:", Remote.ToString());
   Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
   string welcome = "Welcome to my test server";
   data = Encoding.ASCII.GetBytes(welcome);
   newsock.SendTo(data, data.Length, SocketFlags.None, Remote);
   while(true)
   {
     data = new byte[1024];
     recv = newsock.ReceiveFrom(data, ref Remote);
```

```
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
    newsock.SendTo(data, recv, SocketFlags.None, Remote);
   }
  }
}
```

As mentioned, for the UDP server program to accept incoming UDP messages, it must be bound to a specific UDP port on the local system. This is accomplished by creating an IPEndPoint object using the appropriate local IP address (or as shown here, the IPAddress.Any address to use any network interface on the system), and the appropriate UDP port:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
Socket newsock = new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);
newsock.Bind(ipep);
```

The sample UDP server illustrated in the example will accept any incoming UDP packet on port 9050 from the network.

Note There is not an established connection between hosts, so UDP is not picky about where the packet comes from (unlike TCP). However, because of this feature of UDP, when communicating with multiple UDP clients you must be careful to check the transmission point of the received packets.

When creating your sample UDP server, be careful that you don't choose a UDP port that is already in use by another application on your machine. You can monitor the arrangement of applications that are listening on particular UDP ports by using the netstat command at a command prompt.

The netstat -a command displays all the active TCP and UDP sessions on the system. Depending on how many network applications are configured on your system, you might see lots of output here. The first column shows the protocol type, and the second column gives you the hostname or IP address and the port number. In the third column, you get the IP hostname or address and the port number of the remote device, if the port is connected to a

remote device (for TCP). The last column shows the TCP state of TCP connections.

You can select any UDP port that is not shown as being active in your netstat list. Once the SimpleUdpSrvr program is active, you will see an entry for the UDP port you selected, as shown in the preceding netstat output with the 9050 port number:

```
UDP  abednego:9050     *.*
```

Similar to the TCP client/server model, remote devices communicating with UDP must agree on a system to use for sending and receiving data. If both client and server are waiting for data at the same time, both devices will block and not work. To make this arrangement, the SimpleUdpSrvr program follows the following protocol:

- Wait to receive a message from a client
- Send a welcome banner back to the client
- Wait for additional client messages and send them back

These functions are accomplished using the ReceiveFrom() and SendTo() methods of the Socket class.

Because connectionless sockets do not establish a connection, each SendTo() method must include the remote device EndPoint information. It is imperative that you have this information available from the received message. To obtain it, a blank EndPoint object is created and referenced in the ReceiveFrom() method:

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
EndPoint Remote = (EndPoint)(sender);
recv = newsock.ReceiveFrom(data, ref Remote);
```

The Remote object contains the IP information for the remote device that sent the message to the server. This information identifies the incoming data and is also used if the server wants to return a message to the client:

```
string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes(welcome);
newsock.SendTo(data, data.Length, SocketFlags.None, Remote);
```

Warning As is true for the TCP server in lab 9, "Connection-Oriented Sockets," you must always reset the receive data buffer to its full size before the ReceiveFrom() method call occurs, or the Length property will reflect the length of the previously received message. This could cause some unexpected results.

Unlike the TCP server discussed in lab 9, the UDP server cannot be tested without an appropriate UDP client. The next section describes how to create this client.

A UDP Client

The UDP client program is similar to its partner server program.

Because the client does not need to wait on a specific UDP port for incoming data, it does not use the Bind() method. Instead, it employs a random UDP port assigned by the system when the data is sent, and it uses the same port to receive return messages. If you are in a production environment, you might want to specify a set UDP port for the client as well, so that both the server and client programs use the same port numbers.

Warning Be careful if you are using the UDP server and client programs on the same machine. You cannot Bind() the same UDP port number for both programs, or an error will occur. Only one application can bind to a specific port number at a time. Either select a different port number, or let the system choose a random port for the client.

Listing 15.2 shows the SimpleUdpCLient.cs program that demonstrates the fundamentals of a UDP setup.

Listing 15.2: The SimpleUdpClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class SimpleUdpClient
{
  public static void Main()
  {
   byte[] data = new byte[1024];
   string input, stringData;
```

```
  IPEndPoint ipep = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 9050);
  Socket server = new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);
  string welcome = "Hello, are you there?";
  data = Encoding.ASCII.GetBytes(welcome);
  server.SendTo(data, data.Length, SocketFlags.None, ipep);
  IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
  EndPoint Remote = (EndPoint)sender;
  data = new byte[1024];
  int recv = server.ReceiveFrom(data, ref Remote);
  Console.WriteLine("Message received from {0}:", Remote.ToString());
  Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
  while(true)
  {
    input = Console.ReadLine();
    if (input == "exit")
     break;
    server.SendTo(Encoding.ASCII.GetBytes(input), Remote);
    data = new byte[1024];
    recv = server.ReceiveFrom(data, ref Remote);
    stringData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(stringData);
  }
  Console.WriteLine("Stopping client");
  server.Close();
 }
}
```

The UDP client program first defines an IPEndPoint to which the UDP server device will send packets. If you are running the SimpleUdpSrvr program on a remote device, you must enter the appropriate IP address and UDP port number information in the IPEndPoint definition:

```
IPEndPoint ipep = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 9050);
```

The client program sends a quick message to the server machine to introduce itself, then waits for the welcome banner message to be sent back. Because it does not need to accept UDP messages on a specific port number, the client does not bind the Socket object. It will receive the return UDP message on the same port from which it sent the original message.

The SimpleUdpClient program reads the console input and waits for the phrase exit to appear before leaving the while loop. Once the loop is exited, the socket is closed.

Testing the Client and Server Programs

Start the SimpleUdpSrvr program on the assigned server machine you will use for your testing. When the server program starts, it displays a short message:

```
C:\>SimpleUdpSrvr
Waiting for a client...
```

Nothing too fancy here. After the server is started, you can run the SimpleUdpClient program in either a separate command-prompt window on the same machine, or on another machine on the network. (If you use a remote machine, remember to change the 127.0.0.1 IP address to the address of the server machine.)

SimpleUdpClient immediately sends a message to the server, which in turn should send its welcome banner back. On the client, it looks like this:

```
C:\>SimpleUdpClient
Message received from 127.0.0.1:9050:
Welcome to my test server
```

Notice that the EndPoint object of the remote server indicates that it is indeed sending data out from the same UDP port to which it was bound, 9050.

On the server side, you should see the connection message sent by the client:

```
C:\>SimpleUdpSrvr
Waiting for a client...
Message received from 127.0.0.1:1340:
Hello, are you there?
```

Notice that the client, because it was not bound to a specific UDP port, selected a free UDP port to use for the communication. Once the opening protocol has been completed, you can type messages into the client console

and see them sent to the server and displayed, then echoed back to the client and displayed.

Using Connect() in a UDP Client Example

You may be wondering why the UDP client application's ReceiveFrom() and SendTo() methods are so complicated when you are only receiving and sending data to a single UDP server. Well, they don't have to be. As mentioned, the UDP methods are designed to allow the programmer to send UDP packets to any host on the network at any time. Because no prior connection is required for UDP, you must specify the destination host in each SendTo() and ReceiveFrom() method used in the program. If you're planning to send and receive data from only one host, you can take a shortcut.

After the UDP socket is created, you can use the standard Connect() method normally employed in TCP programs to specify the remote UDP server. With that in place, you can then use the Receive() and Send() methods to transfer data to the remote host. The communication still uses UDP packets, and you get to do a lot less work! This technique is demonstrated in the sample UDP client program in Listing16.3.

Listing 15.3: The OddUdpClient.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
class OddUdpClient
{
  public static void Main()
  {
   byte[] data = new byte[1024];
   string input, stringData;
   IPEndPoint ipep = new IPEndPoint(
         IPAddress.Parse("127.0.0.1"), 9050);
   Socket server = new Socket(AddressFamily.InterNetwork,
         SocketType.Dgram, ProtocolType.Udp);
   server.Connect(ipep);
   string welcome = "Hello, are you there?";
```

```
  data = Encoding.ASCII.GetBytes(welcome);
  server.Send(data);
  data = new byte[1024];
  int recv = server.Receive(data);
  Console.WriteLine("Message received from {0}:", ipep.ToString());
  Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
  while(true)
  {
   input = Console.ReadLine();
   if (input == "exit")
    break;
   server.Send(Encoding.ASCII.GetBytes(input));
   data = new byte[1024];
   recv = server.Receive(data);
   stringData = Encoding.ASCII.GetString(data, 0, recv);
   Console.WriteLine(stringData);
  }
  Console.WriteLine("Stopping client");
  server.Close();
 }
}
```

The OddUdpClient program behaves exactly like the SimpleUdpClient program. The only difference is that once the UDP socket is created, it is connected to a specific IPEndPoint:

```
IPEndPoint ipep = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 9050);
Socket server = new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);
server.Connect(ipep);
```

In this case, the Connect() method does not really do what it says. Because the socket is defined as a UDP datagram socket, no actual connection is made, but the socket information is "set" to the IPEndPoint object. All calls to the Send() and Receive() methods are now automatically referenced to the IPEndPoint object. You do not have to use the clunky SendTo() and Receive() methods to handle the data.

You can test the OddUdpClient program by starting the SimpleUdpSrvr as you normally would and using the OddUdpClient program to send messages to it. Things should work just as they did for the SimpleUdpClient program. You can even watch the traffic with WinDump or Analyzer if you don't believe that it's really sending UDP packets! Also, note that each message is sent as a single packet to the server, preserving all message boundaries, just as in the SimpleUdpClient.cs program.

Lab 16: The Domain Name System (DNS)

Part-1

DNS allows the master host database to be split up and distributed among multiple systems on the Internet. DNS uses a hierarchical database approach, creating levels of information that can be split and stored on various systems throughout the Internet.

In addition to splitting up the database, DNS also provides a means for clients to query the database in real time. All a client needs to know is the location of one DNS server (and maybe a backup or two) in the database hierarchy. If the client queries a DNS server with a hostname not stored on the DNS server's local database, the server can query another DNS server that does have the information and forward it to the client.

To implement the DNS concept, a new database and protocol were created to pass DNS information among clients and servers and to enable DNS servers to update one another.

DNS Structure

The structure of a hierarchical database is similar to an organization chart with nodes connected in a treelike manner (that's the hierarchical part). The top node is called the *root*. The root node does not explicitly show up in Internet host addresses, so it is often referred to as the "nameless" node. Multiple categories were created under the root level to divide the database into pieces called *domains*. Each domain contains DNS servers that are responsible for maintaining the database of computer names for that area of the database (that's the distributed part). The diagram in Figure 17.1 shows how the DNS domains are distributed.

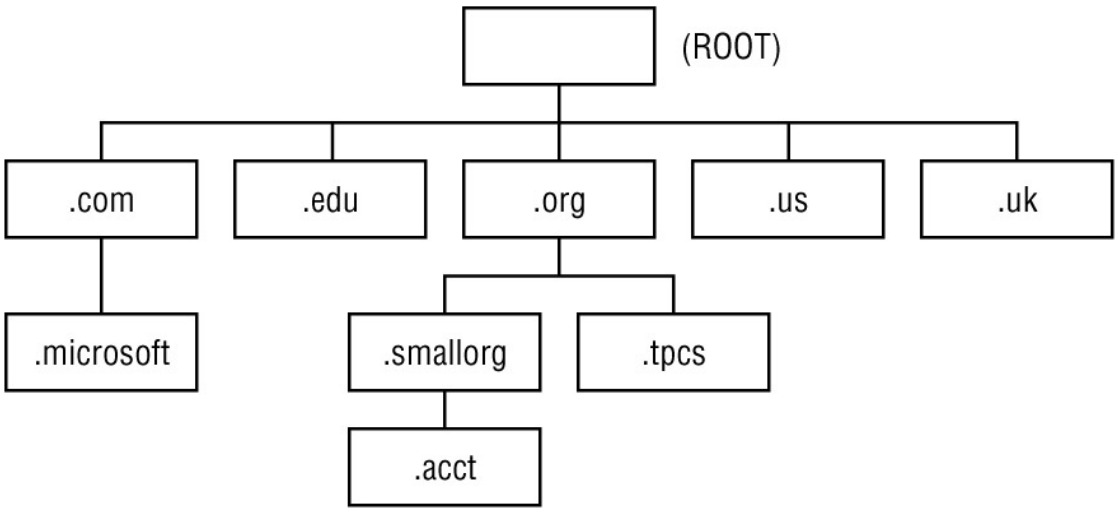


Figure 16.1: The Internet DNS system

The first (or top) level of distribution is divided into domains based on country codes. Additional top-level domains for specific organizations were created to prevent the country domains from getting overcrowded. Table 16.1 describes the layout of the original top-level DNS domains.

Table 16.1: DNS Original Top-Level Domain Names

Domain	**Description**
.com	Commercial organizations
.edu	Educational institutions
.mil	U.S. military sites
.gov	U.S. government organizations
.net	Internet Service Providers (ISPs)
.org	Nonprofit organizations
.us	Other U.S. organizations (such as local governments)
.ca	Canadian organizations
.de	German organizations
(other countries)	Organizations from other countries

Recently, new top-level domains have been added to the DNS system. These new top-level domains allow identification of particular industries worldwide, and not just within a particular country. Corporations within those industries can register hostnames within their particular industry. Table 16.2 shows the seven new top-level domains.

Table 16.2: DNS Top-Level Domains Added in 2001

Domain	**Description**
.aero	Corporations in the air transport industry
.biz	Generic businesses
.coop	Cooperatives
.info	Unrestricted use
.museum	Museums
.name	Individuals
.pro	Professionals (doctors, lawyers, and so on)

As the Internet grows, the top-level domains are each divided into subdomains, or *zones*. Each zone is an independent domain in itself but relies on its parent domain for connectivity to the database. A parent zone must grant permission for a child zone to exist and is responsible for the child zone's behavior (just as in real life). Each zone must have at least two DNS servers that maintain the DNS database for the zone.

The original DNS specifications stipulated that the DNS servers for a single zone must have separate connections to the Internet and be housed in separate locations for fault-tolerance purposes. Because of this stipulation, many organizations rely on other organizations to host their secondary and tertiary DNS servers.

Hosts within a zone add the domain name to their hostname to form a unique Internet name. Thus, computer fred in the smallorg.org domain would be called fred.smallorg.org. This convention can become a little confusing because a domain can contain hosts as well as zones.

For example, the smallorg.org domain can contain host fred.smallorg.org, as well as grant authority for zone acctg.smallorg.org to a subdomain, which in turn can contain another host barney.acctg.smallorg.org. Although this simplifies the database system, it makes finding hosts on the Internet more complicated. Figure 16.2 shows an example of a domain and an associated subdomain.

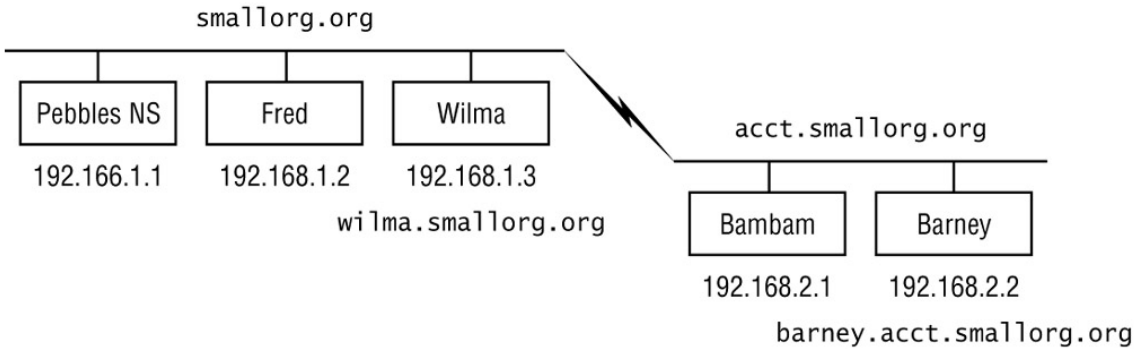


Figure 16.2: A sample domain and subdomain on the Internet

Finding a Hostname in DNS

As mentioned, DNS enables clients to query a local DNS server to obtain hostname information. This process results in three possible scenarios for finding a hostname:

- Finding a host within the local domain
- Finding a remote host whose name is not on the local DNS server
- Finding a remote host whose name is on the local DNS server cache

Local Domain Hostnames

The simplest model for finding a hostname is a client attempting to find the hostname of another system within its local domain. In this scenario, the client uses the local DNS server as its default DNS server and sends the DNS query for the local hostname. The DNS server finds the hostname in its database section and returns the result to the client. This process is demonstrated in Figure 16.3.

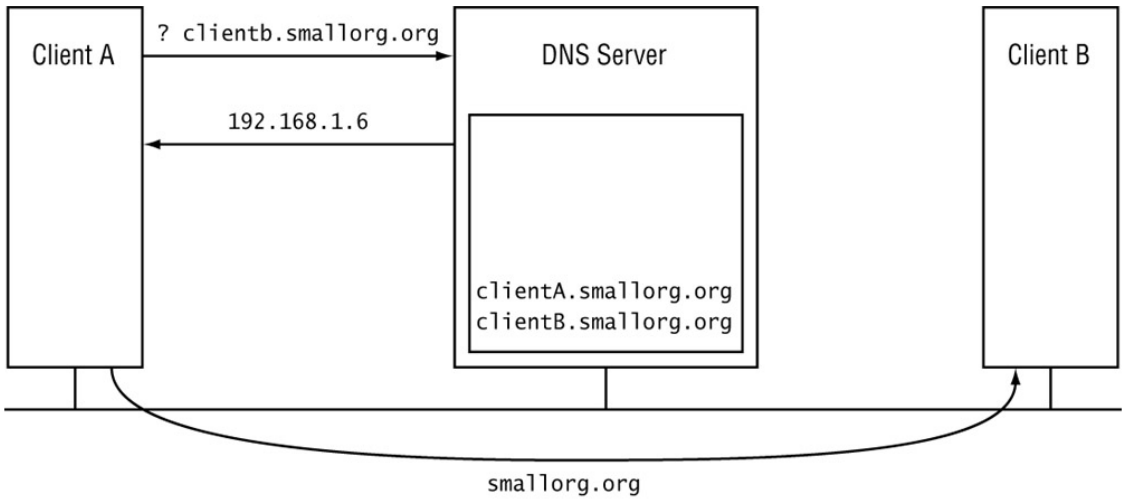


Figure 16.3: Client A is resolving a local hostname from the DNS server

Remote Domain Hostnames

When a client wants to resolve the hostname for a system in a remote domain, there can be two possible scenarios:

- The remote hostname is not found on the local DNS server
- The remote hostname is found on the local DNS server cache

Remote Host Not Found on the Local DNS Server

When the client sends a query to the local DNS server to resolve a hostname for a system not contained within the local DNS server's database, the following actions occur:

1. The local DNS server queries a root DNS server for the hostname.
2. The root DNS server determines what domain the hostname should be found in and passes the request to a DNS server responsible for the host's domain.
3. The responsible local DNS server resolves the hostname to an IP address and returns the result to the root DNS server.
4. The root DNS server passes the result back to the client's local DNS server.
5. The client's local DNS server returns the resulting IP address to the client.

There's a lot of work going on behind the scenes in this process, but the end result is that the client receives the proper IP address for the remote system—all the client had to do was send one query to the local DNS server.

Remote Host Found in the Local DNS Server Cache

One advantage offered by DNS is that when the local DNS server has to query a root DNS server for a remote hostname, it can store the result in a local cache. When a local client queries the local DNS server for a remote hostname about which the server already has information, the local DNS server can return the cached information without having to go out and query a root DNS server. This greatly improves the response time of the query for the client.

There is one caveat to this process. When the root DNS server returns the IP address information for a hostname, it includes a time to live (TTL) value (the period for which the hostname information should be kept in a local DNS server's cache before expiring). The responsible local DNS server for the remote host sets this value. Depending on the volatility of local network hostnames, TTL values can be set anywhere from a few minutes to a few weeks. Once the TTL value has expired, the local DNS server must query the root DNS server when a new client query is received.

The DNS Database

Each DNS server is responsible for keeping track of the hostnames in its zone. To accomplish this, the DNS server must have a way to store host information in a database that can be queried by remote machines. The DNS database is a text file that consists of *resource records* (RRs) for hosts and network functions in the zone. The DNS server must run a DNS server software package to communicate the DNS information from the local domain database to remote DNS servers.

The DNS server's database must define various types of network resources, such as:

- The local domain definition
- Each registered host in the domain
- Common nicknames for hosts in the domain
- Special services, such as DNS servers and mail servers

RR formats were created to track all the information required for the DNS server. Table 16.3 describes some of the basic RRs that a DNS database might contain.

Note DNS database design has become a critical matter for researchers who constantly want to add more information to the database and to control the security of the information that is there. New record types are continually being added to the DNS database, but those in Table 17.3 are the core records for establishing a zone in the DNS database.

Table 16.3: Core DNS Database Records

Record Type	**Description**
SOA	Start of Authority
A	Internet address
NS	Name server
CNAME	Canonical name (nickname)
HINFO	Host information
MX	Mail server
PTR	Pointer (IP address)

There is one SOA record for the domain listed at the top of the database. Any other resource records for the domain can be added in any order after that. Each domain DNS server contains resource records for each registered host in the domain. The following sections describe the individual DNS record types in more detail.

Start of Authority Record (SOA)

Each database starts with an SOA record that defines the zone in which the database resides. The format for the SOA record is as follows:

domain name [TTL] [class] SOA origin person (
serial number
refresh

retry
expire
minimum)

domain name The name of the zone that is being defined. (The @ sign is often used as a placeholder to signify the computer's default domain.)

TTL The length of time (in seconds) for which a requesting computer will keep any DNS information from this zone in its local name cache. Specifying this value is optional.

class The protocol being used (for hosts on the Internet, it will always be class IN, for Internet).

origin The name of the computer where the master zone database is located. A trailing period should be used after the hostname; otherwise, the local domain name will be appended to the hostname (of course, if you want to use that feature, omit the trailing period).

person An e-mail address of a person responsible for the zone. The format of this value is a little different from what you might be used to seeing. The @ sign has already been used to signify the default domain name, so it can't be used in the mail address. Instead, a period is used. For example, instead of sysadm@smallorg.org, you would use sysadm.smallorg.org. If there are any periods in the name part, they must be escaped out by using a backslash \. An example of this would be the address john.jones@smallorg.org, which would translate to john\.jones.smallorg.org.

serial number A unique number that identifies the version of the zone database file. Often what is used here is the date created plus a version count (such as 200210151).

refresh The length of time (in seconds) a secondary DNS server should query a primary DNS server to check the SOA serial number. If the secondary server's SOA serial number is different from the primary server's, the secondary server will request an update to its database. One hour (3,600 seconds) is the common specification for this value.

retry The time (in seconds) after which a secondary DNS server should retry in the event of a failed database refresh attempt.

expire The period (in seconds) for which a secondary DNS server can use the data retrieved from the primary DNS server without getting refreshed. This value will usually be substantial, such as 3600000 (about 42 days).

minimum The length time (in seconds) that should be used as the TTL in all RRs in this zone. Usually 86,400 (1 day) is a good value.

Internet Address Record (A)

Each host in the zone defined by the database should have a valid A record to define its hostname to the Internet. The format for the A record is as follows:

host [TTL] [class] A *address*

host The fully qualified hostname for the computer (including the domain name).

address The IP address of the computer.

TTL and class These parameters are optional and have the same meaning as for the SOA record.

Canonical Name Record (CNAME)

In addition to a normal hostname, many computers also have nicknames. This is useful for identifying particular services without having to rename computers in the domain. For instance, you might assign the nickname www.smallorg.org to the host fred.acctg1.smallorg.org. The CNAME record links nicknames with the real hostname. The format of the CNAME record is as follows:

nickname [TTL] [class] CNAME *hostname*

The roles of the *nickname* and *hostname* parameters are fairly obvious. They represent the nickname assigned and the original hostname of the computer, respectively.

Here again, the *TTL* and *class* parameters are optional and have the same meaning as for the SOA record.

Name Server Record (NS)

Each zone should have at least two DNS servers. NS records are used to identify these servers to other DNS servers trying to resolve hostnames within the zone. The format of an NS record is as follows:

domain [TTL] [class] NS *server*

domain The domain name of the zone for which the DNS server is responsible. If it is blank, the NS record refers to the zone defined in the SOA record.

server The hostname of the DNS server. There should also be an associated A record in the database to identify the IP address of the hostname.

TTL and class Again, these parameters are optional and have the same meaning as for the SOA record.

Host Information Record (HINFO)

Additional information about a computer can be made available to DNS servers by using the HINFO record. The format of the HINFO record is as follows:

host *[TTL] [class]* HINFO *hardware software*

host The hostname of the computer the information applies to.

hardware The type of hardware the computer is using.

software The OS type and version of the computer.

The TTL and class Again, these parameters are optional and have the same meaning as for the SOA record.

Pointer Record (PTR)

In addition to an A record, each computer in the zone should have a PTR record. This allows the DNS server to perform reverse queries from the IP address of the computer. Without the PTR information, remote servers could

not determine the domain name where an IP address is located. The format of a PTR record is as follows:

IN-ADDR name [TTL] [class] PTR *name*

IN-ADDR name The reverse DNS name of the IP address. If that sounds confusing, it is. This name allows the DNS server to work its way backward from the IP address of the computer. The *IN-ADDR.ARPA* address is a special domain to support gateway location and Internet address to host mapping. Inverse queries are not necessary because the IP address is mapped to a fictitious hostname. Thus, the *IN-ADDR name* of a computer with IP address 192.168.0.1 would be 1.0.168.192.IN-ADDR.ARPA.

name The hostname of the computer as found in the A record.

TTL and class Again, these parameters are optional and have the same meaning as for the SOA record.

Mail Exchange Record (MX)

The MX record is used to signify a particular type of host in the domain. It instructs remote mail servers where to forward mail for the domain. The format of the MX record is as follows:

name [TTL] [class] MX *preference host*

name The domain name (or the SOA domain if *name* is blank). This can also be a hostname if you want to redirect mail for a particular host in the network.

preference An integer signifying the order in which remote servers should try connecting if multiple mail servers are specified. The highest preference is 0, with decreasing preference represented by increasing numbers.

preference This feature is used to create primary and secondary mail servers for a domain. When a remote mail server queries the DNS server for a mail server responsible for the domain, the entire list of servers and preferences is sent. The remote mail server should attempt to connect to the highest priority mail server listed, and if that fails, continue down the list in order of preference.

host The hostname or IP address of the mail server. There should also be an associated A record to identify the IP address of the mail server.

TTL and class Again, these parameters are optional and have the same meaning as for the SOA record.

Warning	Be careful about using CNAME entries as MX hosts. Some e-mail server packages cannot work properly with CNAME (nickname) hosts.

A Sample DNS Database

When an ISP hosts a company's domain name, it will have records in its DNS database identifying the domain to the Internet. The SOA record will identify the domain name, but it will point to the ISP's host as the authoritative host. The NS records for the domain will point to the ISP's DNS servers. If the company has its own mail server, the MX record will point to it. If the ISP handles the e-mail server for the company, the MX records point to the ISP's mail servers.

As far as the rest of the Internet is concerned, these computers are part of the company domain—even if they do not really exist on the company network but rather are located at the ISP. Listing 16.1 presents a sample ISP record of the domain definitions in its DNS database.

Listing 16.1: Sample DNS database entries

```
smallorg.org IN SOA  master.isp.net. postmaster.master.isp.net
  postmaster.master.isp.net (
              1999080501  ;unique serial number
             8H    ; refresh rate
             2H    ;retry period
             1W    ; expiration period
             1D)    ; minimum
     NS  ns1.isp.net.  ;defines primary namserver
     NS  ns2.isp.net.  ;defines secondary nameserver
     MX  10 mail1.isp.net.  ; defines primary mail server
     MX  20 mail2.isp.net.  ; defines secondary mail server
 www   CNAME  host1.isp.net  ;defines a www server at the ISP
```

```
ftp   CNAME  host1.isp.net  ; defines an FTP server at the ISP
 host1.isp.net  A  10.0.0.1
1.0.0.10.IN-ADDR.ARPA  PTR  host1.isp.net  ; pointer for reverse DNS
```

The first section of Listing 16.1 is the SOA record for the new domain. The ISP points the domain name smallorg.org to its server master.isp.net. Next, the primary and secondary DNS servers are defined using the NS record type. Following the NS records, the primary (mail1.isp.net) and secondary (mail2.isp.net) mail servers are defined with MX records. Because the preference number for the mail1.isp.net server is lower, it is considered the primary mail server. Any mail for the smallorg.org domain should be delivered to that server if it is available.

After the MX records, the CNAME record defines the hostname www.smallorg.org as a nickname that points to the ISP server, which hosts the company web pages. The address ftp.smallorg.org is also defined as a nickname pointing to the same ISP server, which also hosts the FTP site. Using alias addresses for web and FTP servers is a service that most ISPs provide to customers who cannot afford to have a dedicated connection to the Internet but want to provide web and FTP services to their customers.

The A and PTR recordsets provide the Internet hostname and IP address information for the ISP host so that remote clients can connect to this server.

Note PTR records are often placed in a separate database file on the server to help simplify the databases. This isn't a problem in the Listing 16.1 example, which has just one PTR record, but it can be when there are dozens or hundreds of them.

Lab 17: The Domain Name System (DNS)

Part-2

DNS Classes in C#

The System.Net namespace contains the Dns class, which provides all the necessary DNS functions for C# programs. This section describes the Dns class methods and shows how they can be used in C# programs to utilize the DNS capabilities of the system.

Synchronous Methods

There are four synchronous methods defined in the Dns class:

- •GetHostName()
- •GetHostByName()
- •GetHostByAddress()
- •Resolve()

GetHostName()

The GetHostName() method is used to determine the hostname of the local system the program is running on. This information is frequently needed for network programs, so you'll see this method used a lot. The format is simple: there are no parameters to enter, and the result is a string object:

```
string hostname = Dns.GetHostName();
```

The information retrieved by GetHostName()hostname should be the same name that appears in the Registry Hostname data value, along with the Domain data value, to create the complete fully-qualified domain name (FQDN) of the system. The FQDN includes the local hostname, along with the full domain name information.

GetHostByName()

The GetHostByName() method performs a DNS query for a specified hostname using the default DNS server configured on the system. The format of the method is as follows:

IPHostEntry GetHostByName(string *hostname*)

The IPHostEntry that is returned by GetHostByName() is itself an interesting object. It associates a DNS hostname with an array of alias names and IP addresses. It contains three properties:

- •AddressList: An array of IPAddress objects, one for each assigned IP address
- •Aliases: An array of string objects, one for each alias
- •HostName: A string object for the hostname

The AddressList and Aliases objects must be separated into their individual array elements in order to retrieve the information stored. This is often done using the foreach function in C#.

Listing 17.5 is a sample program that demonstrates the GetHostByName() method.

Listing 17.5: The GetDNSHostInfo.cs program

```
using System;
using System.Net;
class GetDNSHostInfo
{
 public static void Main(string[] argv)
 {
  if (argv.Length != 1)
  {
   Console.WriteLine("Usage: GetDNSHostInfo hostname");
   return;
  }
  IPHostEntry results = Dns.GetHostByName(argv[0]);
  Console.WriteLine("Host name: {0}",
        results.HostName);
  foreach(string alias in results.Aliases)
  {
   Console.WriteLine("Alias: {0}", alias);
  }
  foreach(IPAddress address in results.AddressList)
  {
   Console.WriteLine("Address: {0}",
```

```
            address.ToString());
    }
   }
}
```

The GetDNSHostInfo.cs program is fairly straightforward. It takes the parameter entered on the command line, attempts to use it to perform a GetHostByName(), and dumps the IPHostEntry object returned from the method. If there is more than one IP address assigned to the hostname, each address will appear in the AddressList property. The foreach function is used to extract each individual IP address and display it.

The GetHostByName() method returned the same information that was received by the default nslookup command. In fact, you can run windump or analyzer and watch the DNS query generated by the GetHostByName() method. It should look similar to the one generated by the default nslookup example in Listing 17.4.

The GetHostByName() method is often used with the GetHostName() method to determine the IP address of the local system, like this:

```
IPHostEntry localaddrs = Dns.GetHostByName(Dns.GetHostName());
```

The resulting IPHostEntry object will contain all the IP addresses configured for the local host in the AddressList property.

GetHostByAddress()

When you use the GetDNSHostInfo.cs program in Listing 17.5 and enter an IP address, it unfortunately does not produce the hostname:

```
C:\>GetDNSHostInfo 207.46.197.100
Host name: 207.46.197.100
Address: 207.46.197.100
C:\>
```

Not too exciting. Obviously, the GetHostByName() method only works for resolving hostnames.

When you do need to find the hostname for a known IP address, use the GetHostByAddress() method. There are two formats for this method:

IPHostEntry GetHostByAddress(IPAddress address)
IPHostEntry GetHostByAddress(string address)

The first format is used when you have the IP address as an IPAddress object. The second format is used if you want to use the string representation of the dotted quad format of the IP address (such as 207.46.197.100).

The GetDNSAddressInfo.cs program shown in Listing 17.6 demonstrates the GetHostByAddress() method.

Listing 17.6: The GetDNSAddressInfo.cs program

```
using System;
using System.Net;
class GetDNSAddressInfo
{
  public static void Main(string)[] argv)
  {
   if (argv.Length != 1)
   {
     Console.WriteLine("Usage: GetDNSAddressInfo address");
     return;
   }
   IPAddress test = IPAddress.Parse(argv[0]);
   IPHostEntry iphe = Dns.GetHostByAddress(test);
   Console.WriteLine("Information for {0}",
         test.ToString());
   Console.WriteLine("Host name: {0}", iphe.HostName);
   foreach(string alias in iphe.Aliases)
   {
     Console.WriteLine("Alias: {0}", alias);
   }
   foreach(IPAddress address in iphe.AddressList)
   {
     Console.WriteLine("Address: {0}", address.ToString());
   }
  }
}
```

Just for fun (and experience), this program converts the string argument into an IPAddress object, which is used as the parameter to the GetHostByAddress() method. The output should look something like this:

```
C:\>GetDNSAddressInfo 207.46.197.113
Information for 207.46.197.113
Host name: www.international.microsoft.com
Address: 207.46.197.113
C:\>
```

Here again, you can use windump or analyzer to watch the DNS network traffic generated by this query.

Warning The GetHostByAddress() method does not always work. You may run into many situations where an IP address does not resolve to a hostname. There are two common reasons for this. One is when there is no DNS hostname assigned to the address. Another is that, although a DNS A record might exist, the DNS administrator didn't use a PTR record pointing the address back to the hostname.

Resolve()

As you saw in the preceding sections, the main disadvantage with the GetHostByName() and GetHostByAddress() methods is that they are specific to one or the other types of address information. If you feed an IP address into the GetHostByName() method, it returns only the address. Even worse, if you try to feed a string hostname to the GetHostByAddress() method, it will produce an Exception (because it uses the IPAddress.Parse() method on the supplied string IP address).

Suppose you are creating a program where customers are entering the address of a remote host—you might not know which form the customers will use. Instead of worrying about trying to figure out whether the input is numeric or text so that you can use the proper GetHostBy method, the Dns class offers a simple solution: the Resolve() method. Resolve() accepts an address in either hostname or IP address format and returns the DNS information in an IPHostEntry object. Listing 17.7 demonstrates this method.

Listing 17.7: The GetResolveInfo.cs program

```
using System;
using System.Net;
class GetResolveInfo
{
  public static void Main(string[] argv)
  {
   if (argv.Length != 1)
   {
     Console.WriteLine("Usage: GetResolveInfo address");
     return;
   }
   IPHostEntry iphe = Dns.Resolve(argv[0]);
   Console.WriteLine("Information for {0}", argv[0]);
   Console.WriteLine("Host name: {0}", iphe.HostName);
   foreach(string alias in iphe.Aliases)
   {
     Console.WriteLine("Alias: {0}", alias);
   }
   foreach(IPAddress address in iphe.AddressList)
   {
     Console.WriteLine("Address: {0}",
          address.ToString());
   }
  }
}
```

The Resolve() method attempts to fill the IPHostEntry object with information regarding the address entered as the parameter. You can use either hostnames or IP addresses, and Resolve() will return the proper information. It's a much friendlier way of handling any possible host information that a customer might throw at your program.

Lab 18: Remoting

One of the most exciting features of the .NET Framework is the ability to easily communicate with applications distributed across multiple computers, located on separate networks. *Remoting*, as supported by the .NET Framework, allows applications to share class data and methods among computers on the network, similar to the concept of web services.With remoting, class methods can be hosted on a network device and called from any C# program, without the use of a Microsoft IIS server running on the device.

Remoting relies on data *serialization*.Serialization allows class data elements of any datatype to be transmitted across a stream and reassembled. For remoting, the stream that transmits the data is, of course, a NetworkStream object.

This lab begins by describing serialization and how you can use serialization without remoting to send data between network applications on network devices. Next, you'll examine the elements of .NET remoting and how they interact to provide a distributed computing environment for your C# network programs. Finally, you'll walk through an example of implementing remoting in your network programs

This lab describes two classes that serialize data for transmission across a network, the BinaryFormatter and SoapFormatter classes. These classes can be used to easily convert class instances into a serial stream of bytes that can be sent across the network to a remote system and converted back into the original data.

Using a Serialization Class

There are three steps that are required to serialize a class and send it across the network:

1. Create a library object for the serialized class.
2. Write a sender program that creates instances of the serialized class and send it to a stream.
3. Write a receiver program to read data from the stream and re-create the original serialized class data.

Creating the Serialized Class

Each data class that transports data across the network must be tagged with the [Serializable] attribute in the source code file. This indicates that, by default, all of the data elements in the class will be serialized for transit. Listing 19.1 shows how to create a serialized version of the Employee class.

Listing 19.1: The SerialEmployee.cs program

```
using System;
[Serializable]
public class SerialEmployee
```

```
{
  public int EmployeeID
  public string LastName;
  public string FirstName;
  public int YearsService;
  public double Salary;
  public SerialEmployee()
  {
   EmployeeID = 0;
   LastName = null;
   FirstName = null;
   YearsService = 0;
   Salary = 0.0;
  }
}
```

The SerialEmployee.cs file contains a class definition for the SerialEmployee class. The class contains the data elements used to track basic employee information, along with a simple default constructor for the class. Positioned before the class definition, the [Serializable] attribute indicates that the class can be converted to a serial stream of bytes using one of the formatter classes.

To use this class to transport employee data, you must first create a library file that can be compiled into application programs:

```
csc /t:library SerialEmployee.cs
```

The output from this command will be the SerialEmployee.dll file. This file must be included as a resource in any program that uses the SerialEmployee class.

Warning In serialization, it is extremely important to remember that all applications that use the serialized data class must use the same data library file. When .NET creates the serialized data to send, the data stream includes the class name that defines the data. If the class name does not match when the data is read, the program will not be able to deserialize the stream into the original data class.

Creating a Sender Program

After you create the data class, you can build an application that uses instances of the new class and performs the serialization of the new data to a stream. As mentioned, the BinaryFormatter and SoapFormatter classes serialize the data.

BinaryFormatter serializes the data into a binary stream, much like the GetBytes() method of the Employee.cs program in Listing 7.11. In addition to the actual data, additional information, such as the class name and a version number, are added to the serialized data.

Alternatively, you can use the SoapFormatter class to pass the data using the XML format, similar to the technique used by the web service programs described in Chapter

14. The benefit of using XML is that it is portable between any system or application that recognizes the XML format.

First, you must create an instance of a Stream class to send the data across. This can be any type of stream, including a FileStream, MemoryStream, or of course, a NetworkStream. Next, you create an instance of the appropriate serialization class and use the Serialize() method to send the data across the Stream object:

```
Stream str = new FileStream(
  "testfile.bin", FileMode.Create, FileAccess.ReadWrite);
IFormatter formatter = new BinaryFormatter();
formatter.Serialize(str, data);
```

The IFormatter class creates an instance of the desired serialization class (either BinaryFormatter or SoapFormatter), and the data is serialized using the Serialize() method of the formatter.

To see what information is passed on the network in a serialized data class, you can use a FileStream object to save the output to a file and view the file. Listing 18.2 shows the SoapTest.cs program, which serializes two instances of the SerialEmployee class in a file using the SoapFormatter.

Listing 18.2: The SoapTest.cs program

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
class SoapTest
{
  public static void Main()
  {
   SerialEmployee emp1 = new SerialEmployee();
   SerialEmployee emp2 = new SerialEmployee();
   emp1.EmployeeID = 1;
   emp1.LastName = "Blum";
   emp1.FirstName = "Katie Jane";
   emp1.YearsService = 12;
   emp1.Salary = 35000.50;
   emp2.EmployeeID = 2;
   emp2.LastName = "Blum";
   emp2.FirstName = "Jessica";
   emp2.YearsService = 9;
   emp2.Salary = 23700.30;
   Stream str = new FileStream("soaptest.xml", FileMode.Create,
    FileAccess.ReadWrite);
   IFormatter formatter = new SoapFormatter();
   formatter.Serialize(str, emp1);
   formatter.Serialize(str, emp2);
   str.Close();
  }
```

```
}
```

The SoapFormatter class is found in the System.Runtime.Serialization.Formatters.Soap namespace, so it must be declared with a using statement. If you want to use the BinaryFormatter class instead, that is found in the System.Runtime.Serialization.Formatters.Binary namespace. The IFormatter interface is in the System.Runtime.Serialization namespace, so that must also be included. After creating two instances of the SerialEmployee class, the program generates a FileStream object pointing to a file to store the output in, and then uses the Serialize() method to save the two instances to the stream.

To compile the program, remember to include the SerialEmployee.dll file as a resource:

```
csc /r:SerialEmployee.dll SoapTest.cs
```

After running the SoapTest.exe program, you can examine the soaptest.xml file that is generated (Listing 18.3).

Listing 18.3: The soaptest.xml file

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" Â
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC= Â
"http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV= Â
"http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr= Â
"http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle= Â
"http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:SerialEmployee id="ref-1" xmlns:a1= Â
"http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%3D0
.Â
0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<EmployeeID>1</EmployeeID>
<LastName id="ref-3">Blum</LastName>
<FirstName id="ref-4">Katie Jane</FirstName>
<YearsService>12</YearsService>
<Salary>35000.5</Salary>
</a1:SerialEmployee>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" Â
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC= Â
"http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV= Â
"http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr= Â
"http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle= Â
"http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
```

```
<a1:SerialEmployee id="ref-1" xmlns:a1= Â
"http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%3D0
.Â
0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<EmployeeID>2</EmployeeID>
<LastName id="ref-3">Blum</LastName>
<FirstName id="ref-4">Jessica</FirstName>
<YearsService>9</YearsService>
<Salary>23700.3</Salary>
</a1:SerialEmployee>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

By looking over the soaptest.xml file, you can see how SOAP defines each data element in the serialized class. One important feature of the XML data to notice are the following lines:

```
<a1:SerialEmployee id="ref-1" xmlns:a1= Â
"http://schemas.microsoft.com/clr/assem/SerialEmployee%2C%20Version%
3D0.Â
0.0.0.%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
```

Here, the actual class name for the serialized data class is used within the XML definition data. This is important If the receiving program uses a different class name to define the same data class, it will not match with the XML data read from the stream. The classes must match or the read will fail.

Note As demonstrated in Listing 18.3, though the SoapFormatter adds the ability to communicate class information with other systems, it can greatly increase the amount of data sent in the transmission.

Now that you have seen how the data is serialized, you can write a network application that serializes the class data and sends it to a program running on a remote device. Listing 16.4 shows the BinaryDataSender.cs program, which uses the BinaryFormatter class to send the SerialEmployee data.

Listing 18.4: The BinaryDataSender.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
class BinaryDataSender
{
  public static void Main()
  {
   SerialEmployee emp1 = new SerialEmployee();
   SerialEmployee emp2 = new SerialEmployee();
```

```
    emp1.EmployeeID = 1;
    emp1.LastName = "Blum";
    emp1.FirstName = "Katie Jane";
    emp1.YearsService = 12;
    emp1.Salary = 35000.50;
    emp2.EmployeeID = 2;
    emp2.LastName = "Blum";
    emp2.FirstName = "Jessica";
    emp2.YearsService = 9;
    emp2.Salary = 23700.30;
    TcpClient client = new TcpClient("127.0.0.1", 9050);
    IFormatter formatter = new BinaryFormatter();
    NetworkStream strm = client.GetStream();
    formatter.Serialize(strm, emp1);
    formatter.Serialize(strm, emp2);
    strm.Close();
    client.Close();
  }
}
```

The BinaryDataSender program uses the BinaryFormatter to serialize two instances of the SerialEmployee class and sends it to a remote device specified in the TcpClient object. After the two instances are sent, both the Stream and the TcpClient objects are closed.

Note Because both the BinaryFormatter and SoapFormatter classes require a Stream object to send the serialized data, you must use either a TCP Socket object, or a TcpClient object to send the data. You cannot directly use UDP with the serializers.

Creating a Receiver Program

The third and final step in moving the class data across the network is to build a program that can read the data from the stream and assemble it back into the original class data elements. Again, this is done using either the BinaryFormatter or SoapFormatter classes. Obviously, you must use the same class that was used to serialize the data on the sender.

When the formatter classes deserialize the data, the data elements are extracted into a generic Object object. You must typecast the output to the appropriate class to extract the data elements:

```
IFormatter formatter = new BinaryFormatter();
SerialEmployee emp1 = (SerialEmployee)formatter.Deserialize(str);
```

When the data is received from the stream, it is reassembled into the appropriate data class elements. You should take care to ensure that the proper amount of data is present to reconstruct the data class.

Listing 18.5 shows the BinaryDataRcvr.cs program, which accepts the serialized data from the BinaryDataSender program and re-creates the original data classes.

Listing 18.5: The BinaryDataRcvr.cs program

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
class BinaryDataRcvr
{
  public static void Main()
  {
   TcpListener server = new TcpListener(9050);
   server.Start();
   TcpClient client = server.AcceptTcpClient();
   NetworkStream strm = client.GetStream();
   IFormatter formatter = new BinaryFormatter();
   SerialEmployee emp1 = (SerialEmployee)formatter.Deserialize(strm);
   Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
   Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
   Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
   Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
   Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);
   SerialEmployee emp2 = (SerialEmployee)formatter.Deserialize(strm);
   Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
   Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
   Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
   Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
   Console.WriteLine("emp2.Salary = {0}", emp2.Salary);
   strm.Close();
   server.Stop();
  }
}
```

The BinaryDataRcvr program creates a TcpListener object bound to a specific port and waits for a connection attempt from the BinaryDataSender program. When the connection is established, the Deserialize() method converts the received data stream back into the original data class.

Note This example used simple datatypes, but you can easily modify it to add more complex datatypes, such as an employee start date. Just add the new data elements to the SerialEmployee.cs file, re-create the DLL library file, and rebuild the sender and receiver programs.

Problems with Serialization

While the serializing examples so far show a simple technique for serializing and transmitting complex data classes, in the real world, on real networks, it is not often this easy. You may have noticed that the BinaryDataSender and BinaryDataRcvr programs assumed one important thing: they both expected all of the data to arrive at the receiver

for the BinaryFormatter to deserialize the stream into the original class data. As you probably know by now, this is not necessarily what occurs on a real network.

If not all of the data is received on the stream before the Deserialize() method is performed, there will be a problem. When the Deserialize() method does not have enough bytes to complete the reassembly, it produces an Exception, and the data class is not properly created. The solution to this is to use a hybrid technique, combining the serialization classes presented here with the data sizing methods demonstrated back in Chapter 7. If you send the size of each serialized data object before the actual object, the receiver can determine how many bytes of data to receive before attempting to deserialize the data.

The easiest way to do this is to serialize the data coming from the stream into a MemoryStream object. The MemoryStream object holds all of the serialized data in a memory buffer. It allows you to easily determine the total size of the serialized data. When the size of the data stream is determined, both the size value and the serialized data buffer can be sent out the NetworkStream to the remote device.

Listing 18.6 shows the BetterDataSender.cs program, which uses this technique to transmit two instances of the SerialEmployee data class to a remote device.

Listing 18.6: The BetterDataSender.cs program

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
class BetterDataSender
{
  public void SendData (NetworkStream strm, SerialEmployee emp)
  {
   IFormatter formatter = new SoapFormatter();
   MemoryStream memstrm = new MemoryStream();
   formatter.Serialize(memstrm, emp);
   byte[] data = memstrm.GetBuffer();
   int memsize = (int)memstrm.Length;
   byte[] size = BitConverter.GetBytes(memsize);
   strm.Write(size, 0, 4);
   strm.Write(data, 0, memsize);
   strm.Flush();
   memstrm.Close();
  }
  public BetterDataSender()
  {
   SerialEmployee emp1 = new SerialEmployee();
   SerialEmployee emp2 = new SerialEmployee();
   emp1.EmployeeID = 1;
   emp1.LastName = "Blum";
   emp1.FirstName = "Katie Jane";
```

```
    emp1.YearsService = 12;
    emp1.Salary = 35000.50;
    emp2.EmployeeID = 2;
    emp2.LastName = "Blum";
    emp2.FirstName = "Jessica";
    emp2.YearsService = 9;
    emp2.Salary = 23700.30;
    TcpClient client = new TcpClient("127.0.0.1", 9050);
    NetworkStream strm = client.GetStream();
    SendData(strm, emp1);
    SendData(strm, emp2);
    strm.Close();
    client.Close();
  }
  public static void Main()
  {
   BetterDataSender bds = new BetterDataSender();
  }
}
```

The BetterDataSender program uses the SendData() method to create a MemoryStream object with the serialized data to send to the remote device. From the MemoryStream object, the size of the data is determined and sent to the remote receiver. Then the serialized data MemoryStream buffer is sent to the remote receiver.

Now take a look at the BetterDataRcvr.cs program, Listing 18.7, which demonstrates how to reassemble the received data size and serialized data from the sender.

Listing 18.7: The BetterDataRcvr.cs program

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
class BetterDataRcvr
{
  private SerialEmployee RecvData (NetworkStream strm)
  {
   MemoryStream memstrm = new MemoryStream();
   byte[] data = new byte[4];
   int recv = strm.Read(data, 0, 4);
   int size = BitConverter.ToInt32(data, 0);
   int offset = 0;
   while(size > 0)
   {
     data = new byte[1024];
     recv = strm.Read(data, 0, size);
     memstrm.Write(data, offset, recv);
     offset += recv;
```

```
      size -= recv;
    }
    IFormatter formatter = new SoapFormatter();
    memstrm.Position = 0;
    SerialEmployee emp = (SerialEmployee)formatter.Deserialize(memstrm);
    memstrm.Close();
    return emp;
   }
   public BetterDataRcvr()
   {
    TcpListener server = new TcpListener(9050);
    server.Start();
    TcpClient client = server.AcceptTcpClient();
    NetworkStream strm = client.GetStream();
    SerialEmployee emp1 = RecvData(strm);
    Console.WriteLine("emp1.EmployeeID = {0}", emp1.EmployeeID);
    Console.WriteLine("emp1.LastName = {0}", emp1.LastName);
    Console.WriteLine("emp1.FirstName = {0}", emp1.FirstName);
    Console.WriteLine("emp1.YearsService = {0}", emp1.YearsService);
    Console.WriteLine("emp1.Salary = {0}\n", emp1.Salary);
    SerialEmployee emp2 = RecvData(strm);
    Console.WriteLine("emp2.EmployeeID = {0}", emp2.EmployeeID);
    Console.WriteLine("emp2.LastName = {0}", emp2.LastName);
    Console.WriteLine("emp2.FirstName = {0}", emp2.FirstName);
    Console.WriteLine("emp2.YearsService = {0}", emp2.YearsService);
    Console.WriteLine("emp2.Salary = {0}", emp2.Salary);
    strm.Close();
    server.Stop();
   }
   public static void Main()
   {
    BetterDataRcvr bdr = new BetterDataRcvr();
   }
}
```

The BetterDataRcvr program uses the RecvData() method to obtain the incoming data from the sender. First, a 4-byte size value come in. This value indicates how many bytes to expect in the serialized data portion of the transmission. The RecvData() method then goes into a loop until all of the expected bytes are received from the NetworkStream. As the bytes arrive, they are added to a MemoryStream object. When all of the bytes have been received, the MemoryStream object creates the data class instance.

Warning It is important to remember that the MemoryStream object must be reset to point to the start of the stream before being used in the Deserialize() method.

To test the BetterDataSender and BetterDataRcvr programs, you must compile them with the same SerialEmployee.dll file.

Note The BinaryFormatter and SoapFormatter take care of any network byte order issues when transferring the data to the remote host.