PART I
The C# Language

1

Introducing C#

WHAT YOU WILL LEARN IN THIS CHAPTER

- What the .NET Framework is and what it contains
- How .NET applications work
- What C# is and how it relates to the .NET Framework
- What tools are available for creating .NET applications with C#

Welcome to the first chapter of the first section of this book. This section will provide you with the basic knowledge you need to get up and running with C#. This chapter provides an overview of C# and the .NET Framework, including what these technologies are, the motivation for using them, and how they relate to each other.

First is a general discussion of the .NET Framework. This technology contains many concepts that are tricky to come to grips with initially. This means that the discussion, by necessity, covers many new concepts in a short amount of space. However, a quick look at the basics is essential to understanding how to program in C#. Later in the book you will revisit many of the topics covered here, exploring them in more detail.

After that general introduction, the chapter provides a basic description of C# itself, including its origins and similarities to C++. Finally, you look at the primary tools used throughout this book: Visual Studio 2010 (VS) and Visual C# 2010 Express (VCE).

WHAT IS THE .NET FRAMEWORK?

The .NET Framework (now at version 4) is a revolutionary platform created by Microsoft for developing applications. The most interesting thing about this statement is how vague it is — but there are good reasons for this. For a start, note that it doesn't "develop applications

on the Windows operating system." Although the Microsoft release of the .NET Framework runs on the Windows operating system, it is possible to find alternative versions that will work on other systems. One example of this is Mono, an open-source version of the .NET Framework (including a C# compiler) that runs on several operating systems, including various flavors of Linux and Mac OS. In addition, you can use the Microsoft .NET Compact Framework (essentially a subset of the full .NET Framework) on personal digital assistant (PDA) class devices and even some smartphones. One of the key motivations behind the .NET Framework is its intended use as a means of integrating disparate operating systems.

In addition, the preceding definition of the .NET Framework includes no restriction on the type of applications that are possible. That's because there is no restriction — the .NET Framework enables the creation of Windows applications, Web applications, Web services, and pretty much anything else you can think of. Also, with Web applications it's worth noting that these are, by definition, multi-platform applications, since any system with a Web browser can access them. With the recent addition of Silverlight, this category also includes applications that run inside browsers on the client, as well as applications that merely render Web content in the form of HTML.

The .NET Framework has been designed so that it can be used from any language, including C# (the subject of this book) as well as C++, Visual Basic, JScript, and even older languages such as COBOL. For this to work, .NET-specific versions of these languages have also appeared, and more are being released all the time. Not only do all of these have access to the .NET Framework, but they can also communicate with each other. It is perfectly possible for C# developers to make use of code written by Visual Basic programmers, and vice versa.

All of this provides an extremely high level of versatility and is part of what makes using the .NET Framework such an attractive prospect.

What's in the .NET Framework?

The .NET Framework consists primarily of a gigantic library of code that you use from your client languages (such as C#) using object-oriented programming (OOP) techniques. This library is categorized into different modules — you use portions of it depending on the results you want to achieve. For example, one module contains the building blocks for Windows applications, another for network programming, and another for Web development. Some modules are divided into more specific submodules, such as a module for building Web services within the module for Web development.

The intention is for different operating systems to support some or all of these modules, depending on their characteristics. A PDA, for example, would include support for all the core .NET functionality but is unlikely to require some of the more esoteric modules.

Part of the .NET Framework library defines some basic *types*. A type is a representation of data, and specifying some of the most fundamental of these (such as "a 32-bit signed integer") facilitates interoperability between languages using the .NET Framework. This is called the *Common Type System (CTS)*.

As well as supplying this library, the .Net Framework also includes the .NET *Common Language Runtime (CLR)*, which is responsible for maintaining the execution of all applications developed using the .NET library.

Writing Applications Using the .NET Framework

Writing an application using the .NET Framework means writing code (using any of the languages that support the Framework) using the .NET code library. In this book you use VS and VCE for your development. VS is a powerful, integrated development environment that supports C# (as well as managed and unmanaged C++, Visual Basic, and some others). VCE is a slimmed down (and free) version of VS that supports C# only. The advantage of these environments is the ease with which .NET features can be integrated into your code. The code that you create will be entirely C# but use the .NET Framework throughout, and you'll make use of the additional tools in VS and VCE where necessary.

In order for C# code to execute, it must be converted into a language that the target operating system understands, known as *native code*. This conversion is called *compiling* code, an act that is performed by a *compiler*. Under the .NET Framework, this is a two-stage process.

CIL and JIT

When you compile code that uses the .NET Framework library, you don't immediately create operating-system-specific native code. Instead, you compile your code into *Common Intermediate Language (CIL)* code. This code isn't specific to any operating system (OS) and isn't specific to C#. Other .NET languages — Visual Basic .NET, for example — also compile to this language as a first stage. This compilation step is carried out by VS or VCE when you develop C# applications.

Obviously, more work is necessary to execute an application. That is the job of a *just-in-time* (JIT) compiler, which compiles CIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The *just-in-time* part of the name reflects the fact that CIL code is compiled only when it is needed.

In the past, it was often necessary to compile your code into several applications, each of which targeted a specific operating system and CPU architecture. Typically, this was a form of optimization (to get code to run faster on an AMD chipset, for example), but at times it was critical (for applications to work in both Win9*x* and WinNT/2000 environments, for example). This is now unnecessary, because JIT compilers (as their name suggests) use CIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the appropriate one is used to create the native code required.

The beauty of all this is that it requires a lot less work on your part — in fact, you can forget about system-dependent details and concentrate on the more interesting functionality of your code.

NOTE *You may come across references to Microsoft Intermediate Language (MSIL) or just IL. MSIL was the original name for CIL, and many developers still use this terminology.*

Assemblies

When you compile an application, the CIL code created is stored in an *assembly*. Assemblies include both executable application files that you can run directly from Windows without the need for any

other programs (these have a `.exe` file extension) and libraries (which have a `.dll` extension) for use by other applications.

In addition to containing CIL, assemblies also include *meta* information (that is, information about the information contained in the assembly, also known as *metadata*) and optional *resources* (additional data used by the CIL, such as sound files and pictures). The meta information enables assemblies to be fully self-descriptive. You need no other information to use an assembly, meaning you avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, you can just run an executable file from this directory and (assuming the .NET CLR is installed) you're good to go.

Of course, you won't necessarily want to include everything required to run an application in one place. You might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the *global assembly cache (GAC)*. Placing code in the GAC is simple — you just place the assembly containing the code in the directory containing this cache.

Managed Code

The role of the CLR doesn't end after you have compiled your code to CIL and a JIT compiler has compiled that to native code. Code written using the .NET Framework is *managed* when it is executed (a stage usually referred to as *runtime*). This means that the CLR looks after your applications by managing memory, handling security, allowing cross-language debugging, and so on. By contrast, applications that do not run under the control of the CLR are said to be *unmanaged,* and certain languages such as C++ can be used to write such applications, which, for example, access low-level functions of the operating system. However, in C# you can write only code that runs in a managed environment. You will make use of the managed features of the CLR and allow .NET itself to handle any interaction with the operating system.

Garbage Collection

One of the most important features of managed code is the concept of *garbage collection*. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use. Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of your computer followed by a system crash.

.NET garbage collection works by periodically inspecting the memory of your computer and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but you can rest assured that it will happen.

There are some implications for programmers here. Because this work is done for you at an unpredictable time, applications have to be designed with this in mind. Code that requires a lot of memory to run should tidy itself up, rather than wait for garbage collection to happen, but that isn't as tricky as it sounds.

Fitting It Together

Before moving on, let's summarize the steps required to create a .NET application as discussed previously:

1. Application code is written using a .NET-compatible language such as C# (see Figure 1-1).

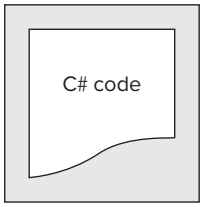

FIGURE 1-1

2. That code is compiled into CIL, which is stored in an assembly (see Figure 1-2).

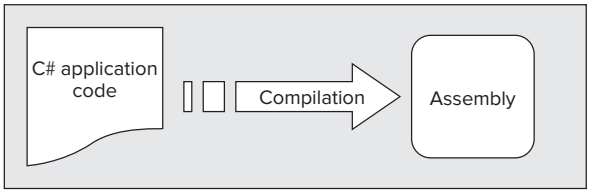

FIGURE 1-2

3. When this code is executed (either in its own right if it is an executable or when it is used from other code), it must first be compiled into native code using a JIT compiler (see Figure 1-3).

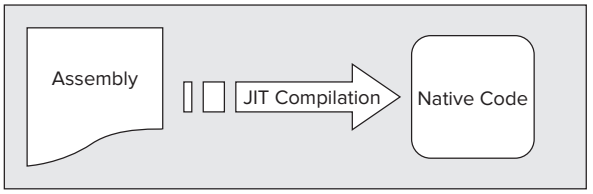

FIGURE 1-3

4. The native code is executed in the context of the managed CLR, along with any other running applications or processes, as shown in Figure 1-4.

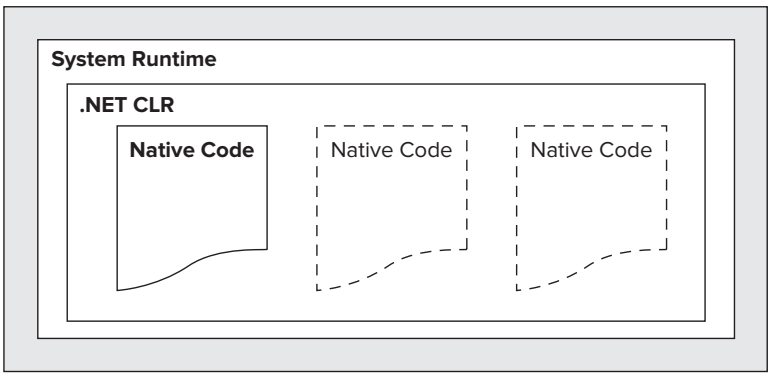

FIGURE 1-4

Linking

Note one additional point concerning this process. The C# code that compiles into CIL in step 2 needn't be contained in a single file. It's possible to split application code across multiple source code files, which are then compiled together into a single assembly. This extremely useful process is known as *linking*. It is required because it is far easier to work with several smaller files than one enormous one. You can separate out logically related code into an individual file so that it can be worked on independently and then practically forgotten about when completed. This also makes it easy to locate specific pieces of code when you need them and enables teams of developers to divide the programming burden into manageable chunks, whereby individuals can "check out" pieces of code to work on without risking damage to otherwise satisfactory sections or sections other people are working on.

WHAT IS C#?

C#, as mentioned earlier, is one of the languages you can use to create applications that will run in the .NET CLR. It is an evolution of the C and C++ languages and has been created by Microsoft specifically to work with the .NET platform. The C# language has been designed to incorporate many of the best features from other languages, while clearing up their problems.

Developing applications using C# is simpler than using C++, because the language syntax is simpler. Still, C# is a powerful language, and there is little you might want to do in C++ that you can't do in C#. Having said that, those features of C# that parallel the more advanced features of C++, such as directly accessing and manipulating system memory, can be carried out only by using code marked as *unsafe*. This advanced programmatic technique is potentially dangerous (hence its name) because it is possible to overwrite system-critical blocks of memory with potentially catastrophic results. For this reason, and others, this book does not cover that topic.

At times, C# code is slightly more verbose than C++. This is a consequence of C# being a *type-safe* language (unlike C++). In layperson's terms, this means that once some data has been assigned to a type, it cannot subsequently transform itself into another unrelated type. Consequently, strict rules must be adhered to when converting between types, which means you will often need to write more code to carry out the same task in C# than you might write in C++. However, you get two benefits: the code is more robust and debugging is simpler, and .NET can always track the type of a piece of data at any time. In C#, you therefore may not be able to do things such as "take the region of memory 4 bytes into this data and 10 bytes long and interpret it as X," but that's not necessarily a bad thing.

C# is just one of the languages available for .NET development, but it is certainly the best. It has the advantage of being the only language designed from the ground up for the .NET Framework and is the principal language used in versions of .NET that are ported to other operating systems. To keep languages such as the .NET version of Visual Basic as similar as possible to their predecessors yet compliant with the CLR, certain features of the .NET code library are not fully supported, or at least require unusual syntax. By contrast, C# can make use of every feature that the .NET Framework code library has to offer. The latest version of .NET includes several additions to the C# language, partly in response to requests from developers, making it even more powerful.

Applications You Can Write with C#

The .NET Framework has no restrictions on the types of applications that are possible, as discussed earlier. C# uses the framework and therefore has no restrictions on possible applications. However, here are a few of the more common application types:

- **Windows applications:** Applications, such as Microsoft Office, that have a familiar Windows look and feel about them. This is made simple by using the Windows Forms module of the .NET Framework, which is a library of *controls* (such as buttons, toolbars, menus, and so on) that you can use to build a Windows user interface (UI). Alternatively, you can use Windows Presentation Foundation (WPF) to build Windows applications, which gives you much greater flexibility and power.
- **Web applications:** Web pages such as those that might be viewed through any Web browser. The .NET Framework includes a powerful system for generating Web content dynamically, enabling personalization, security, and much more. This system is called ASP.NET (Active Server Pages .NET), and you can use C# to create ASP.NET applications using Web Forms. You can also write applications that run inside the browser with Silverlight.
- **Web services:** An exciting way to create versatile distributed applications. Using Web services you can exchange virtually any data over the Internet, using the same simple syntax regardless of the language used to create a Web service or the system on which it resides. For more advanced capabilities, you can also create Windows Communication Foundation (WCF) services.

Any of these types may also require some form of database access, which can be achieved using the ADO.NET (Active Data Objects .NET) section of the .NET Framework, through the ADO.NET Entity Framework, or through the LINQ (Language Integrated Query) capabilities of C#. Many other resources can be drawn on, such as tools for creating networking components, outputting graphics, performing complex mathematical tasks, and so on.

C# in This Book

The first part of this book deals with the syntax and usage of the C# language without too much emphasis on the .NET Framework. This is necessary because you won't be able to use the .NET Framework at all without a firm grounding in C# programming. We'll start off even simpler, in fact, and leave the more involved topic of OOP until you've covered the basics. These are taught from first principles, assuming no programming knowledge at all.

After that, you'll be ready to move on to developing more complex (but more useful) applications. Part II of this book looks at Windows Forms programming, Part III tackles Web application and Web service programming, Part IV examines data access (for database, file system, and XML data), and Part V covers some other .NET topics of interest.

VISUAL STUDIO 2010

In this book, you use the Visual Studio 2010 (VS) or Visual C# 2010 Express (VCE) development tools for all of your C# programming, from simple command-line applications to more complex project types. A development tool, or integrated development environment (IDE), such as VS isn't essential for developing C# applications, but it makes things much easier. You can (if you want to) manipulate C# source code files in a basic text editor, such as the ubiquitous Notepad application, and compile code into assemblies using the command-line compiler that is part of the .NET Framework. However, why do this when you have the power of an IDE to help you?

The following is a short list of some Visual Studio features that make it an appealing choice for .NET development:

- VS automates the steps required to compile source code but at the same time gives you complete control over any options used should you wish to override them.
- The VS text editor is tailored to the languages VS supports (including C#) so that it can intelligently detect errors and suggest code where appropriate as you are typing. This feature is called *IntelliSense*.
- VS includes designers for Windows Forms, Web Forms, and other applications, enabling simple drag-and-drop design of UI elements.
- Many types of C# projects may be created with "boilerplate" code already in place. Instead of starting from scratch, you will often find that various code files are started for you, reducing the amount of time spent getting started on a project. This is especially true of the "Starter Kit" project type, which enables you to develop from a fully functional application base. Some starter kits are included with the VS installation, and you can find plenty more online to play with.
- VS includes several wizards that automate common tasks, many of which can add appropriate code to existing files without you having to worry about (or even, in some cases, remember) the correct syntax.
- VS contains many powerful tools for visualizing and navigating through elements of your projects, whether they are C# source code files or other resources such as bitmap images or sound files.

- As well as simply writing applications in VS, you can create deployment projects, making it easy to supply code to clients and for them to install it without much trouble.
- VS enables you to use advanced debugging techniques when developing projects, such as the capability to step through code one instruction at a time while keeping an eye on the state of your application.

There is much more than this, but you get the idea!

Visual Studio 2010 Express Products

In addition to Visual Studio 2010, Microsoft also supplies several simpler development tools known as Visual Studio 2010 Express Products. These are freely available at `http://www.microsoft.com/express`.

Two of these products, Visual C# 2010 Express and Visual Web Developer 2010 Express, together enable you to create almost any C# application you might need. They both function as slimmed-down versions of VS and retain the same look and feel. While they offer many of the same features as VS, some notable feature are absent, although not so many that they would prevent you from using these tools to work through the chapters.

In this book you'll use VCE to develop C# applications wherever possible, and only use VS where it is necessary for certain functionality. Of course, if you have VS there is no need to use an express product.

Solutions

When you use VS or VCE to develop applications, you do so by creating *solutions*. A solution, in VS and VCE terms, is more than just an application. Solutions contain *projects,* which might be Windows Forms projects, Web Form projects, and so on. Because solutions can contain multiple projects, you can group together related code in one place, even if it will eventually compile to multiple assemblies in various places on your hard disk.

This is very useful because it enables you to work on shared code (which might be placed in the GAC) at the same time as applications that use this code. Debugging code is a lot easier when only one development environment is used, because you can step through instructions in multiple code modules.

SUMMARY

In this chapter, you looked at the .NET Framework in general terms and discovered how it makes it easy for you to create powerful and versatile applications. You saw what is necessary to turn code in languages such as C# into working applications, and what benefits you gain from using managed code running in the .NET CLR.

You also learned what C# actually is and how it relates to the .NET Framework, and you were introduced to the tools that you'll use for C# development — Visual Studio 2010 and Visual C# 2010 Express.

In the next chapter, you get some C# code running, which will give you enough knowledge to sit back and concentrate on the C# language itself, rather than worry too much about how the IDE works.

► WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
.NET Framework fundamentals	The .NET Framework is Microsoft's latest development platform, and is currently in version 4. It includes a common type system (CTS) and common language runtime (CLR). .NET Framework applications are written using object oriented programming (OOP) methodology, and usually contain managed code. Memory management of managed code is handled by the .NET runtime; this includes garbage collection.
.NET Framework applications	Applications written using the .NET framework are first compiled into CIL. When an application is executed, the JIT compiles this CIL into native code. Applications are compiled and different parts are linked together into assemblies that contain the CIL.
C# basics	C# is one of the languages included in the .NET Framework. It is an evolution of previous languages such as C++, and can be used to write any number of applications, including both web sites and Windows applications.
Integrated Development Environments (IDEs)	You can use Visual Studio 2010 to write any type of .NET application using C#. You can also use the free, but less powerful, express product range (including Visual C# Developer Express) to create .NET applications in C#. Both of these IDEs work with solutions, which can consist of multiple projects.

2

Writing a C# Program

WHAT YOU WILL LEARN IN THIS CHAPTER

- A basic working knowledge of Visual Studio 2010 and Visual C# 2010 Express Edition
- How to write a simple console application
- How to write a Windows Forms application

Now that you've spent some time learning what C# is and how it fits into the .NET Framework, it's time to get your hands dirty and write some code. You use Visual Studio 2010 (VS) and Visual C# 2010 Express (VCE) throughout this book, so the first thing to do is have a look at some of the basics of these development environments.

VS is an enormous and complicated product, and it can be daunting to first-time users, but using it to create basic applications can be surprisingly simple. As you start to use VS in this chapter, you will see that you don't need to know a huge amount about it to begin playing with C# code. Later in the book you'll see some of the more complicated operations that VS can perform, but for now a basic working knowledge is all that is required.

VCE is far simpler for getting started, and in the early stages of this book all the examples are described in the context of this IDE. However, if you prefer, you can use VS instead, and everything will work in more or less the same way. For that reason, you'll see both IDEs in this chapter, starting with VS.

After you've had a look at the IDEs, you put together two simple applications. You don't need to worry too much about the code in these for now; you just prove that things work. By working through the application creation procedures in these early examples, they will become second nature before too long.

The first application you create is a simple *console application*. Console applications are those that don't make use of the graphical windows environment, so you won't have to worry about

buttons, menus, interaction with the mouse pointer, and so on. Instead, you run the application in a command prompt window and interact with it in a much simpler way.

The second application is a *Windows Forms application*. The look and feel of this is very familiar to Windows users, and (surprisingly) the application doesn't require much more effort to create. However, the syntax of the code required is more complicated, even though in many cases you don't actually have to worry about details.

You use both types of application over the next two parts of the book, with slightly more emphasis on console applications at the beginning. The additional flexibility of Windows applications isn't necessary when you are learning the C# language, while the simplicity of console applications enables you to concentrate on learning the syntax and not worry about the look and feel of the application.

THE DEVELOPMENT ENVIRONMENTS

This section explores the VS and VCE development environments, starting with VS. These environments are similar, and you should read both sections regardless of which IDE you are using.

Visual Studio 2010

When VS is first loaded, it immediately presents you with a host of windows, most of which are empty, along with an array of menu items and toolbar icons. You will be using most of these in the course of this book, and you can rest assured that they will look far more familiar before too long.

If this is the first time you have run VS, you will be presented with a list of preferences intended for users who have experience with previous releases of this development environment. The choices you make here affect a number of things, such as the layout of windows, the way that console windows run, and so on. Therefore, choose Visual C# Development Settings; otherwise, you may find that things don't quite work as described in this book. Note that the options available vary depending on the options you chose when installing VS, but as long as you chose to install C# this option will be available.

If this isn't the first time that you've run VS, but you chose a different option the first time, don't panic. To reset the settings to Visual C# Development Settings you simply have to import them. To do this, select Tools ➪ Import and Export Settings, and choose the Reset All Settings option, shown in Figure 2-1.

Click Next, and indicate whether you want to save your existing settings before proceeding. If you have customized things, you might want to do this; otherwise, select No and click Next again. From the next dialog, select Visual C# Development Settings, shown in Figure 2-2. Again, the available options may vary.

Finally, click Finish to apply the settings.

The VS environment layout is completely customizable, but the default is fine here. With C# Developer Settings selected, it is arranged as shown in Figure 2-3.

FIGURE 2-1

FIGURE 2-2

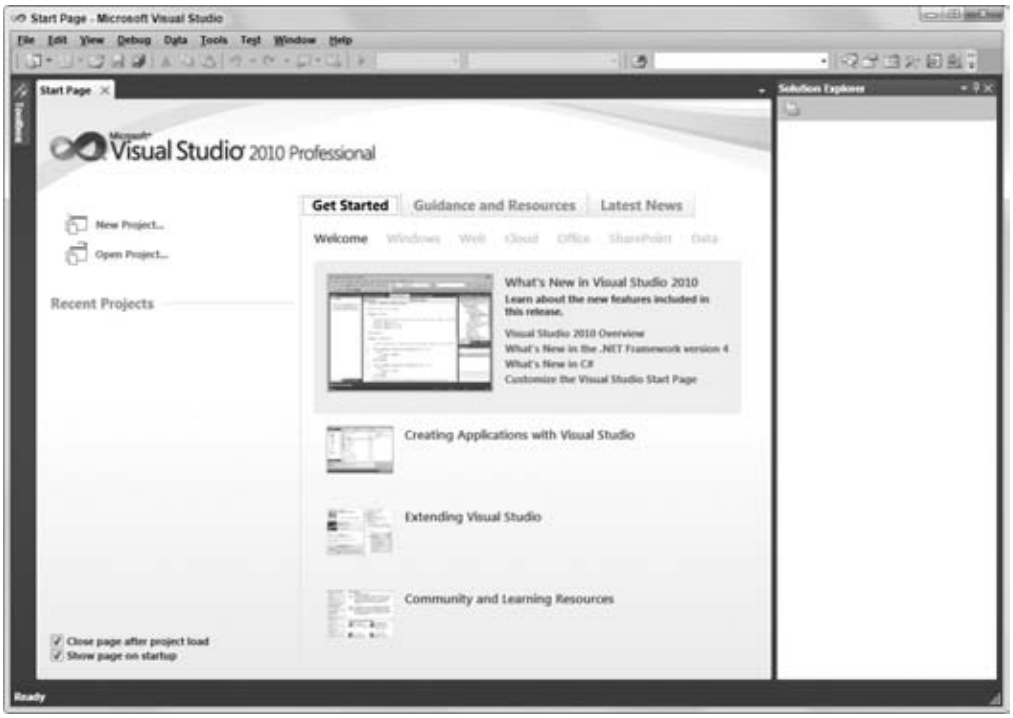

FIGURE 2-3

The main window, which contains a helpful Start Page by default when VS is started, is where all your code is displayed. This window can contain many documents, each indicated by a tab, so you can easily switch between several files by clicking their filenames. It also has other functions: It can display GUIs that you are designing for your projects, plain-text files, HTML, and various tools that are built into VS. You will come across all of these in the course of this book.

Above the main window are toolbars and the VS menu. Several different toolbars can be placed here, with functionality ranging from saving and loading files to building and running projects to debugging controls. Again, you are introduced to these as you need to use them.

Here are brief descriptions of each of the main features that you will use the most:

- The Toolbox toolbar pops up when the mouse moves over it. It provides access to, among other things, the user interface building blocks for Windows applications. Another tab, Server Explorer, can also appear here (selectable via the View Server Explorer menu option) and includes various additional capabilities, such as providing access to data sources, server settings, services, and more.
- The Solution Explorer window displays information about the currently loaded *solution*. A solution, as you learned in the previous chapter, is VS terminology for one or more projects along with their configurations. The Solution Explorer window displays various views of the projects in a solution, such as what files they contain and what is contained in those files.
- Just below the Solution Explorer window you can display a Properties window, not shown in Figure 2-3 because it appears only when you are working on a project (you can also toggle its

display using View Properties Window). This window provides a more detailed view of the project's contents, enabling you to perform additional configuration of individual elements. For example, you can use this window to change the appearance of a button in a Windows form.

- Also not shown in the screenshot is another extremely important window: the Error List window, which you can display using View Error List. It shows errors, warnings, and other project-related information. The window updates continuously, although some information appears only when a project is compiled.

This may seem like a lot to take in, but it doesn't take long to get used to. You start by building the first of your example projects, which involves many of the VS elements just described.

NOTE *VS is capable of displaying many other windows, both informational and functional. Many of these can share screen space with the windows mentioned here, and you can switch between them using tabs. Several of these windows are used later in the book, and you'll probably discover more yourself when you explore the VS environment in more detail.*

Visual C# 2010 Express Edition

With VCE you don't have to worry about changing the settings. Obviously, this product isn't going to be used for Visual Basic programming, so there is no equivalent setting to worry about here. When you start VCE for the first time, you are presented with a screen that is very similar to the one in VS (see Figure 2-4).

FIGURE 2-4

CONSOLE APPLICATIONS

You use console applications regularly in this book, particularly at the beginning, so the following Try It Out provides a step-by-step guide to creating a simple one. It includes instructions for both VS and VCE.

TRY IT OUT Creating a Simple Console Application

1. Create a new console application project by selecting File ➪ New ➪ Project in VS or File ➪ New Project in VCE, as shown in Figures 2-5 and 2-6.

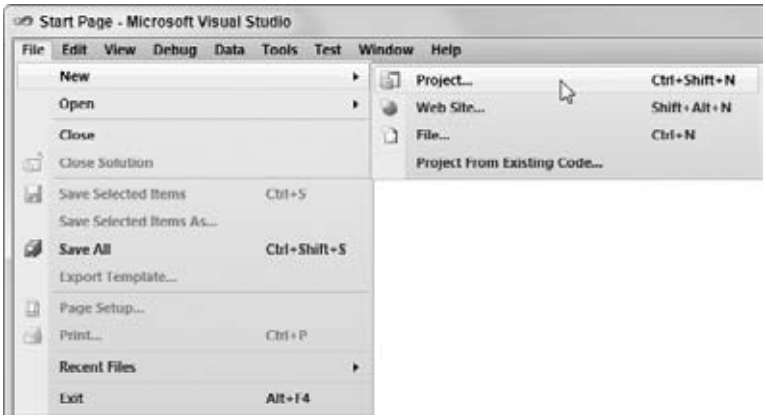

FIGURE 2-5

2. In VS, ensure that the Visual C# node is selected in the Installed Templates pane of the window that appears, and choose the Console Application project type in the middle pane (see Figure 2-7). In VCE, simply select Console Application in the Templates pane (see Figure 2-8). In VS, change the Location text box to C:\BegVCSharp\Chapter02 (this directory is created automatically if it doesn't already exist). For both VS and VCE, leave the default text in the Name text box (ConsoleApplication1) and the other settings as they are (refer to Figures 2-7 and 2-8).
3. Click the OK button.
4. If you are using VCE, after the project is initialized click the Save All button on the toolbar or select Save All from the File menu, set the Location field to C:\BegVCSharp\Chapter02, and click Save.

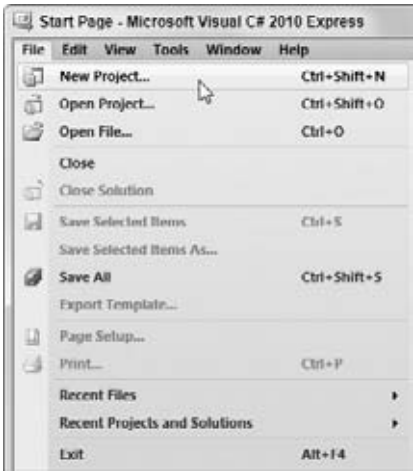

FIGURE 2-6

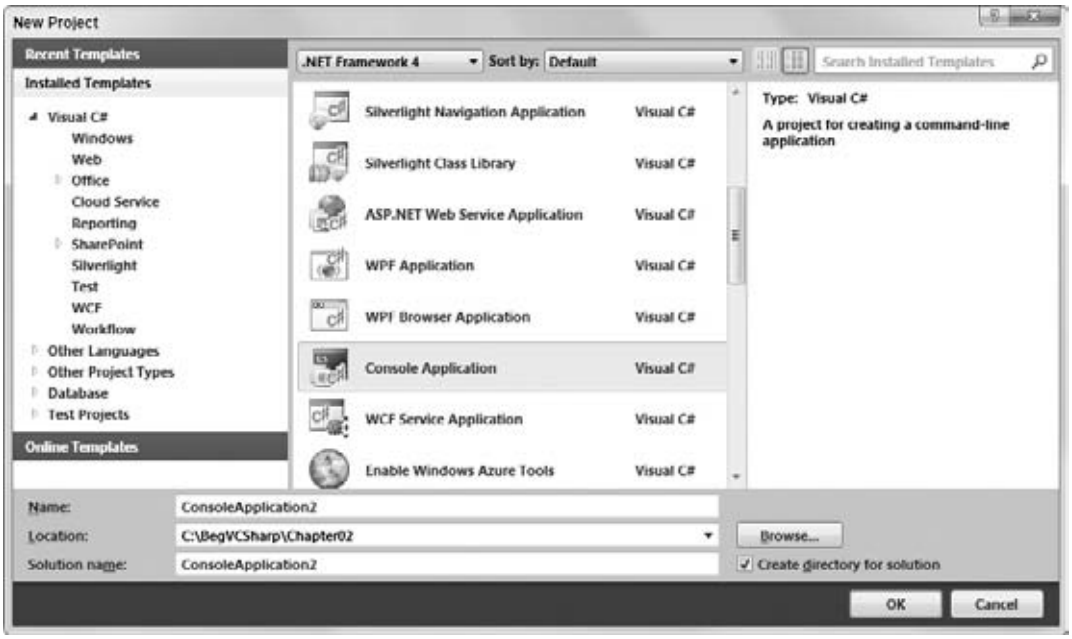

FIGURE 2-7

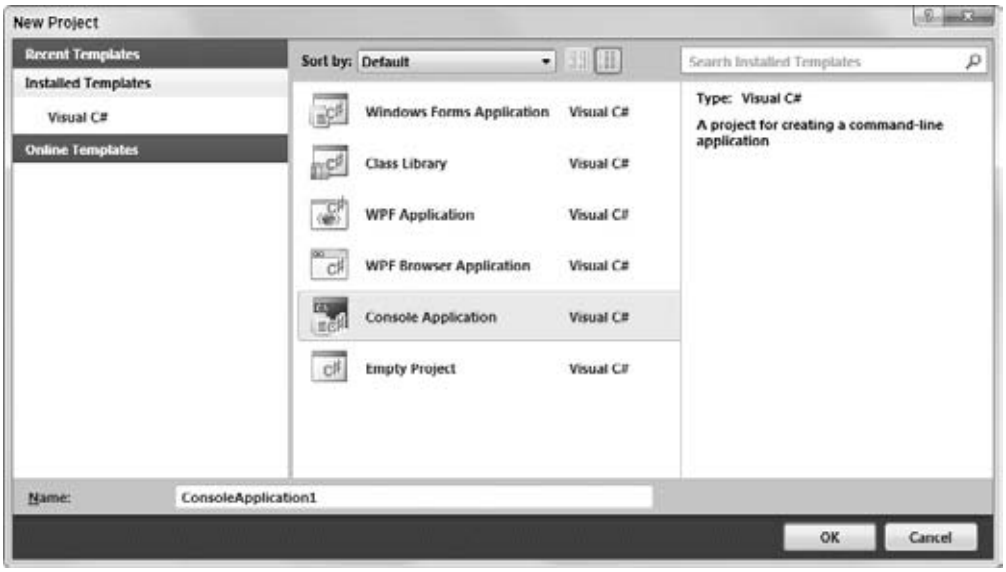

FIGURE 2-8

5. Once the project is initialized, add the following lines of code to the file displayed in the main window:


```
namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // Output text to the screen.
         Console.WriteLine("The first app in Beginning C# Programming!");
         Console.ReadKey();
      }
   }
}
```

Code snippet ConsoleApplication1\Program.cs

6. Select the Debug ➪ Start Debugging menu item. After a few moments you should see the window shown in Figure 2-9.

FIGURE 2-9

7. Press any key to exit the application (you may need to click on the console window to focus on it first).

In VS, the preceding display appears only if the Visual C# Developer Settings are applied, as described earlier in this chapter. For example, with Visual Basic Developer Settings applied, an empty console window is displayed, and the application output appears in a window labeled Immediate. In this case, the `Console.ReadKey()` code also fails, and you see an error. If you experience this problem, the best solution for working through the examples in this book is to apply the Visual C# Developer Settings — that way, the results you see match the results shown here. If this problem persists, then open the Tools ➪ Options dialog and uncheck the Debugging ➪ Redirect all Output . . . option, as shown in Figure 2-10.

How It Works

For now, we won't dissect the code used thus far because the focus here is on how to use the development tools to get code up and running. Clearly, both VS and VCE do a lot of the work for you and make the process of compiling and executing code simple. In fact, there are multiple ways to perform even these basic steps — for instance, you can create a new project by using the menu item mentioned earlier, by pressing Ctrl+Shift+N, or by clicking the corresponding icon in the toolbar.

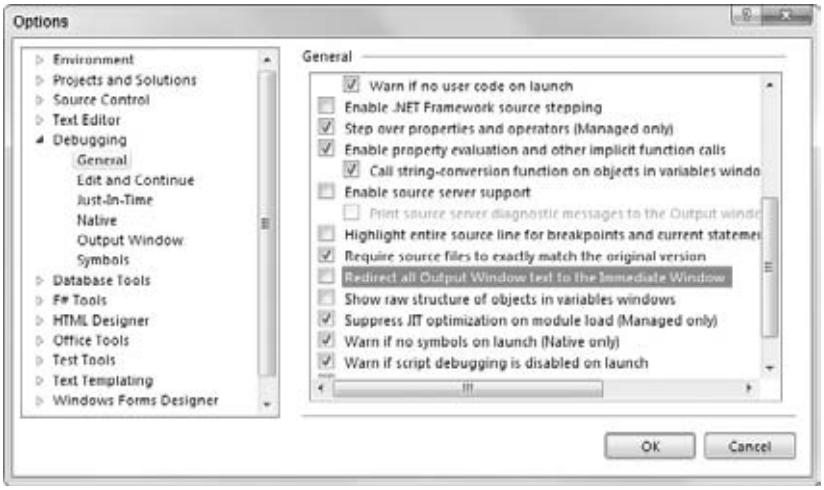

FIGURE 2-10

Similarly, your code can be compiled and executed in several ways. The process you used previously — selecting Debug ➪ Start Debugging — also has a keyboard shortcut (F5) and a toolbar icon. You can also run code without being in debugging mode using the Debug ➪ Start Without Debugging menu item (or by pressing Ctrl+F5), or compile your project without running it (with debugging on or off) using Build ➪ Build Solution or F6. Note that you can execute a project without debugging or build a project using toolbar icons, although these icons don't appear on the toolbar by default. After you have compiled your code, you can also execute it simply by running the `.exe` file produced in Windows Explorer, or from the command prompt. To do this, open a command prompt window, change the directory to `C:\BegVCSharp\Chapter02\ConsoleApplication1\ConsoleApplication1\bin\Debug\`, type **ConsoleApplication1**, and press Enter.

NOTE *In future examples, when you see the instructions "create a new console project" or "execute the code," you can choose whichever method you want to perform these steps. Unless otherwise stated, all code should be run with debugging enabled. In addition, the terms "start," "execute," and "run" are used interchangeably in this book, and discussions following examples always assume that you have exited the application in the example.*

Console applications terminate as soon as they finish execution, which can mean that you don't get a chance to see the results if you run them directly through the IDE. To get around this in the preceding example, the code is told to wait for a key press before terminating, using the following line:

```
Console.ReadKey();
```

You will see this technique used many times in later examples. Now that you've created a project, you can take a more detailed look at some of the regions of the development environment.

The Solution Explorer

The Solution Explorer window is in the top-right corner of the screen. It is the same for both VS and VCE (as are all the windows examined in this chapter unless otherwise specified). By default, this window is set to auto-hide, but you can dock it to the side of the screen by clicking the pin icon when it is visible. The Solution Explorer window shares space with another useful window called Class View, which you can display using View ➪ Class View. Figure 2-11 shows both of these windows with all nodes expanded (you can toggle between them by clicking on the tabs at the bottom of the window when the window is docked).

NOTE *In VCE, the Class View window is only available if you turn on Expert Settings, which you can do through the Tools ➪ Settings ➪ Expert Settings menu item.*

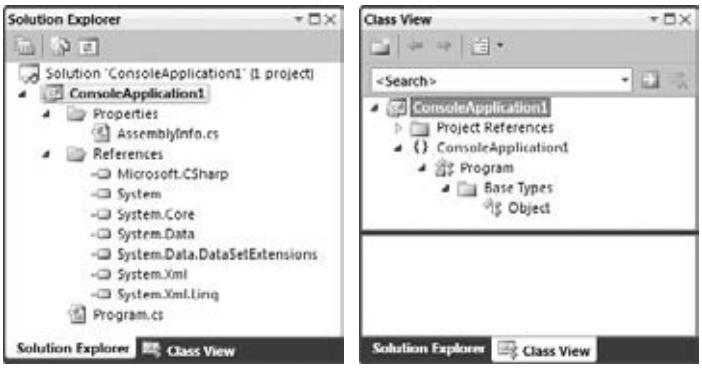

FIGURE 2-11

This Solution Explorer view shows the files that make up the ConsoleApplication1 project. The file to which you added code, `Program.cs`, is shown along with another code file, `AssemblyInfo.cs`, and several references.

NOTE *All C# code files have a* `.cs` *file extension.*

You don't have to worry about the `AssemblyInfo.cs` file for the moment. It contains extra information about your project that doesn't concern us yet.

You can use this window to change what code is displayed in the main window by double-clicking `.cs` files; right-clicking them and selecting View Code; or by selecting them and clicking the toolbar button that appears at the top of the window. You can also perform other operations on files here, such as renaming them or deleting them from your project. Other file types can also appear here, such as project resources

(resources are files used by the project that might not be C# files, such as bitmap images and sound files). Again, you can manipulate them through the same interface.

The `References` entry contains a list of the .NET libraries you are using in your project. You'll look at this later; the standard references are fine for now. The other view, Class View, presents an alternative view of your project by showing the structure of the code you created. You'll come back to this later in the book; for now the Solution Explorer display is appropriate. As you click on files or other icons in these windows, you may notice that the contents of the Properties window (shown in Figure 2-12) changes.

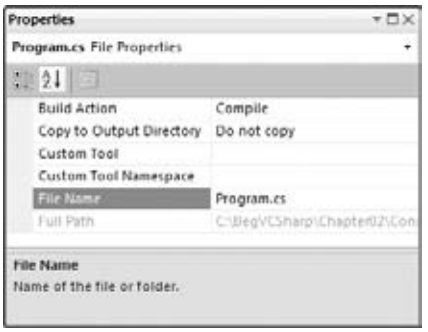

FIGURE 2-12

The Properties Window

The Properties window (select View ➪ Properties Window if it isn't already displayed) shows additional information about whatever you select in the window above it. For example, the view shown in Figure 2-12 is displayed when the `Program.cs` file from the project is selected. This window also displays information about other selected items, such as user interface components (as shown in the "Windows Forms Applications" section of this chapter).

Often, changes you make to entries in the Properties window affect your code directly, adding lines of code or changing what you have in your files. With some projects, you spend as much time manipulating things through this window as making manual code changes.

The Error List Window

Currently, the Error List window (View ➪ Error List) isn't showing much of interest because there is nothing wrong with the application. However, this is a very useful window indeed. As a test, remove the semicolon from one of the lines of code you added in the previous section. After a moment, you should see a display like the one shown in Figure 2-13.

FIGURE 2-13

In addition, the project will no longer compile.

NOTE *In Chapter 3, when you start looking at C# syntax, you will learn that semicolons are expected throughout your code — at the end of most lines, in fact.*

This window helps you eradicate bugs in your code because it keeps track of what you have to do to compile projects. If you double-click the error shown here, the cursor jumps to the position of the error in your source code (the source file containing the error will be opened if it isn't already open), so you can fix it quickly. Red wavy lines appear at the positions of errors in the code, so you can quickly scan the source code to see where problems lie.

The error location is specified as a line number. By default, line numbers aren't displayed in the VS text editor, but that is something well worth turning on. To do so, tick the Line numbers check box in the Options dialog (selected via the Tools ➪ Options menu item). It appears in the Text Editor ➪ C# ➪ General category, as shown in Figure 2-14.

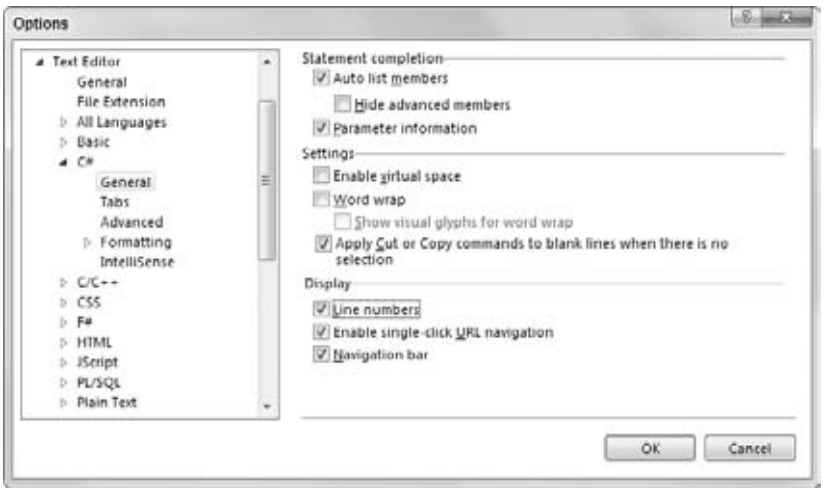

FIGURE 2-14

NOTE *In VCE you must select Show All Settings for this option to become available, and the list of options looks slightly different from Figure 2-14.*

Many useful options can be found through this dialog, and you will use several of them later in this book.

WINDOWS FORMS APPLICATIONS

It is often easier to demonstrate code by running it as part of a Windows application than through a console window or via a command prompt. You can do this using user interface building blocks to piece together a user interface.

The following Try It Out shows just the basics of doing this, and you'll see how to get a Windows application up and running without a lot of details about what the application is actually doing. Later you take a detailed look at Windows applications.

TRY IT OUT Creating a Simple Windows Application

1. Create a new project of type Windows Forms Application (VS or VCE) in the same location as before (`C:\BegVCSharp\Chapter02`; and if you are using VCE, save the project to this location after you create it) with the default name WindowsFormsApplication1. If you are using VS and the first project is still open, make sure the Create New Solution option is selected to start a new solution. These settings are shown in Figures 2-15 and 2-16.

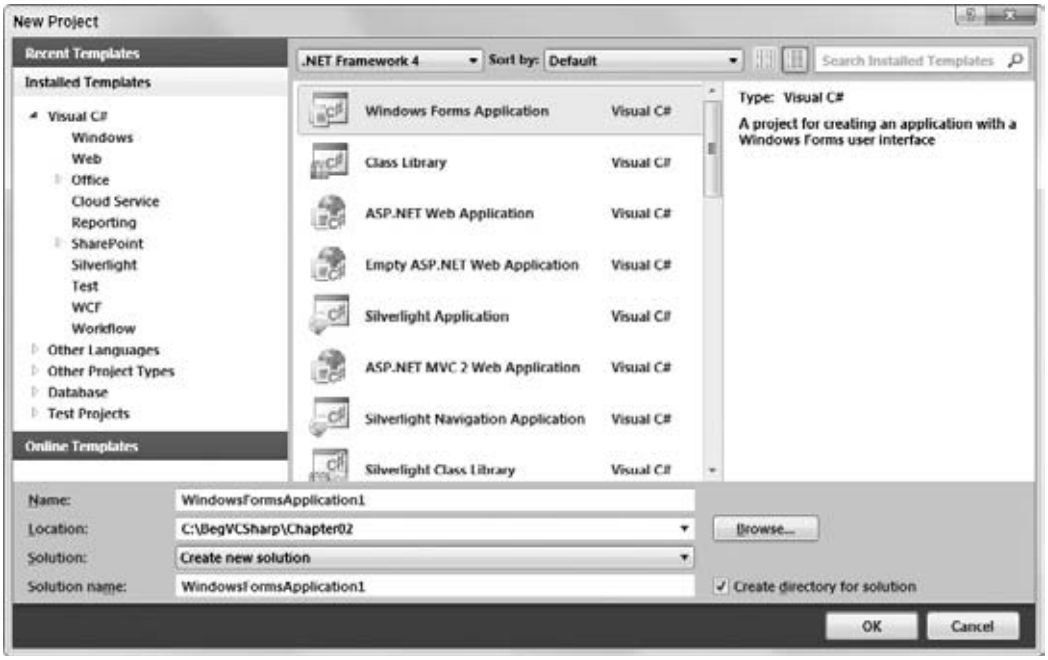

FIGURE 2-15

2. Click OK to create the project. You should see an empty Windows form. Move the mouse pointer to the Toolbox bar on the left of the screen, then to the Button entry of the All Windows Forms tab, and double-click the entry to add a button to the main form of the application (Form1).
3. Double-click the button that has been added to the form.
4. The C# code in `Form1.cs` should now be displayed. Modify it as follows (only part of the code in the file is shown here for brevity):


```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("The first Windows app in the book!");
}
```

Code snippet WindowsFormsApplication1\Form1.cs

5. Run the application.

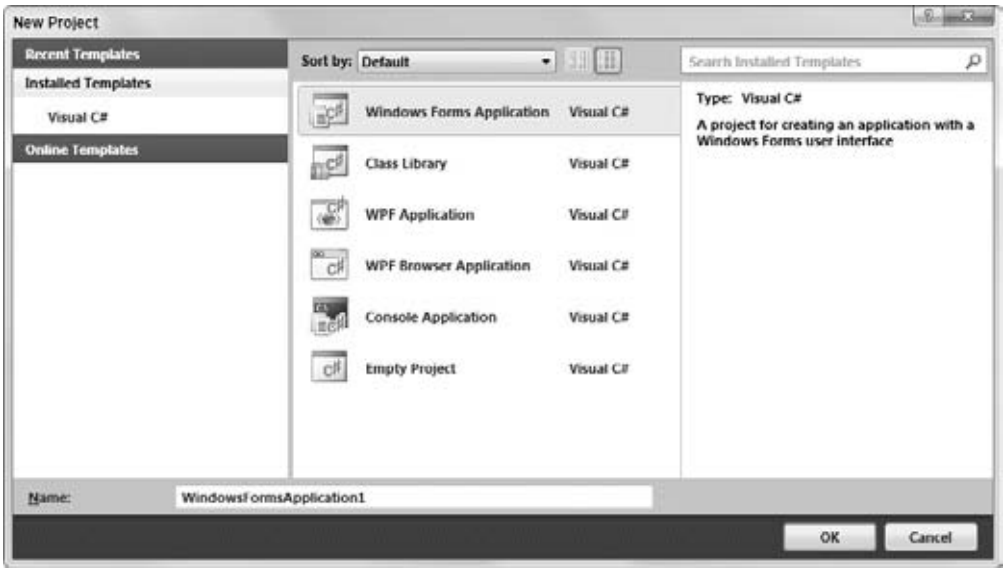

FIGURE 2-16

6. Click the button presented to open a message dialog box, as shown in Figure 2-17.

7. Click OK, and then exit the application by clicking the X in the top-right corner, as per standard Windows applications.

FIGURE 2-17

How It Works

Again, it is plain that the IDE has done a lot of work for you and made it simple to create a functional Windows application with little effort. The application you created behaves just like other windows — you can move it around, resize it, minimize it, and so on. You don't have to write the code to do that — it works. The same is true for the button you added. Simply by double-clicking it, the IDE knew that you wanted to write code to execute when a user clicked the button in the running application. All you had to do was provide that code, getting full button-clicking functionality for free.

Of course, Windows applications aren't limited to plain forms with buttons. Look at the toolbar where you found the Button option and you'll see a whole host of user interface building blocks, some of which may be familiar. You will use most of these at some point in the book, and you'll find that they are all easy to use, saving you a lot of time and effort.

The code for your application, in `Form1.cs`, doesn't look much more complicated than the code in the previous section, and the same is true for the code in the other files in the Solution Explorer window. Much of the code generated is hidden by default. It is concerned with the layout of controls on the form,

which is why you can view the code in design view in the main window — it's a visual translation of this layout code. A button is an example of a control that you can use, as are the rest of the UI building blocks found in the Windows Forms section of the Toolbox bar.

You can take a closer look at the button as a control example. Switch back to the design view of the form using the tab on the main window, and click once on the button to select it. When you do so, the Properties window in the bottom-right corner of the screen shows the properties of the button control (controls have properties much like the files shown in the last example). Ensure that the application isn't currently running, scroll down to the `Text` property, which is currently set to `button1`, and change the value to `Click Me`, as shown in Figure 2-18.

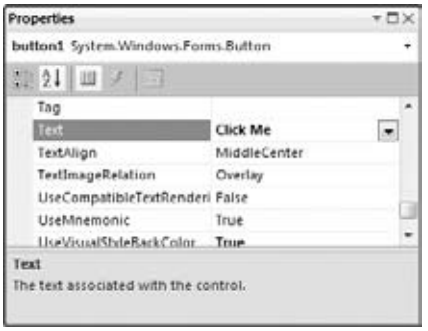

FIGURE 2-18

The text written on the button in Form1 should also reflect this change.

There are many properties for this button, ranging from simple formatting of the color and size to more obscure settings such as data binding settings, which enable you to establish links to databases. As briefly mentioned in the previous example, changing properties often results in direct changes to code, and this is no exception. However, if you switch back to the code view of `Form1.cs`, you won't see any change in the code.

To see the modified code, you need to look at the hidden code mentioned previously. To view the file that contains this code, expand `Form1.cs` in the Solution Explorer, which reveals a `Form1.Designer.cs` node. Double-click that file to see what's inside.

At a cursory glance, you might not notice anything in this code reflecting the button property change at all. That's because the sections of C# code that deal with the layout and formatting of controls on a form are hidden (after all, you hardly need to look at the code if you have a graphical display of the results).

VS and VCE use a system of *code outlining* to achieve this subterfuge. You can see this in Figure 2-19.

Looking down the left side of the code (just next to the line numbers if you've turned them on), you may notice some gray lines and boxes with + and – symbols in them. These boxes are used to expand and contract regions of code. Toward the bottom of the file is a box with a + in it, and a box in the main body of the code reading "Windows Form Designer generated code." This label is basically saying, "Here is some code generated by VS that you don't need to know about." You can look at it if you want, however, and see what you have done by changing the button properties. Simply click the box with the + in it and the code will become visible, and somewhere within it you should see the following line:

```
this.button1.Text = "Click Me";
```

Without worrying too much about the syntax used here, you can see that the text you typed in the Properties window has popped up directly in your code.

This outlining method can be very handy when you are writing code because you can expand and contract many other regions, not just those that are normally hidden. Just as looking at a book's table of contents can help you by providing a quick summary of what it contains, looking at a series of collapsed regions of code can make it much easier to navigate through what can be vast amounts of C# code.

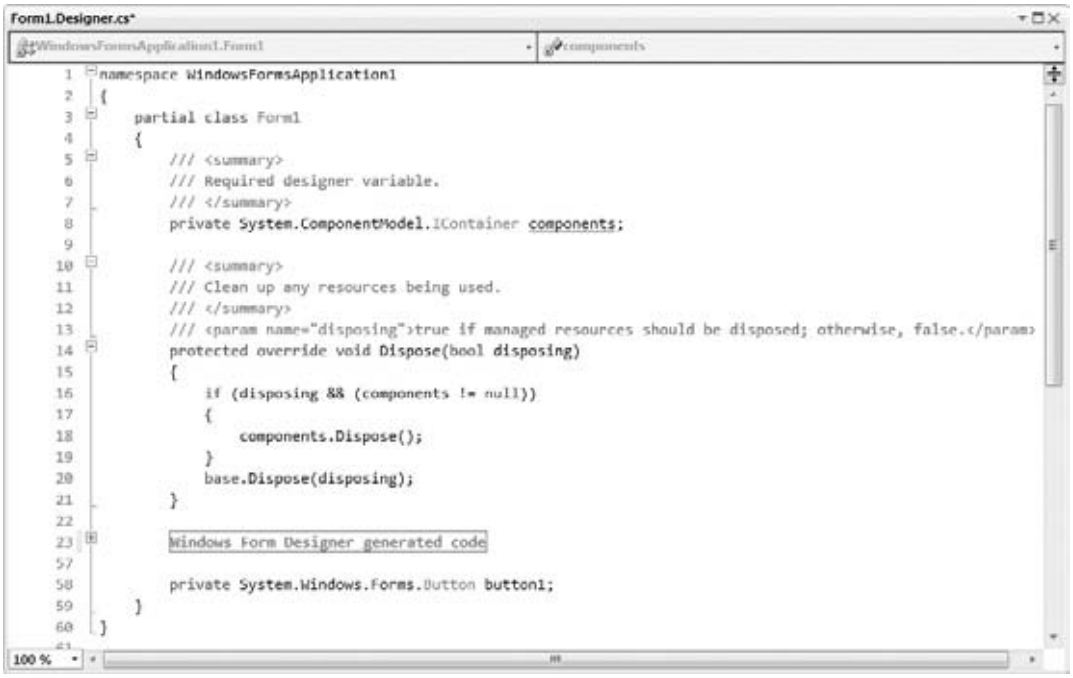

FIGURE 2-19

SUMMARY

This chapter introduced some of the tools that you will use throughout the rest of this book. You have had a quick tour around the Visual Studio 2010 and Visual C# 2010 Express development environments and used them to build two types of applications. The simpler of these, the console application, is quite enough for most needs, and it enables you to focus on the basics of C# programming. Windows applications are more complicated but are visually more impressive and intuitive to use for anyone accustomed to a Windows environment (and let's face it, that's most of us).

Now that you know how to create simple applications, you can get down to the real task of learning C#. After dealing with basic C# syntax and program structure, you move on to more advanced object-oriented methods. Once you've covered all that, you can begin to learn how to use C# to gain access to the power available in the .NET Framework.

For subsequent chapters, unless otherwise specified, instructions refer to VCE, although, as shown in this chapter, adapting these instructions for VS is not difficult, and you can use whichever IDE you prefer, or to which you have access.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Visual Studio 2010 settings	This book requires the C# development settings option, chosen when first run or by resetting settings.
Console applications	Console applications are simple command-line applications, used in much of this book to illustrate techniques. Create a new console application with the Console Application template that you see when you create a new project in VS or VCE. To run a project in debug mode, use the Debug ➪ Start Debugging menu item, or press F5.
IDE windows	The project contents are shown in the Solution Explorer window. The properties of the selected item are shown in the Properties window. Errors are shown in the Error List window.
Windows Forms applications	Windows Forms applications are applications that have the look and feel of standard desktop applications, including the familiar icons to maximize, minimize, and close an application. They are created with the Windows Forms template in the New Project dialog box.

3

Variables and Expressions

WHAT YOU WILL LEARN IN THIS CHAPTER

- Basic C# syntax
- Variables and how to use them
- Expressions and how to use them

To use C# effectively, it's important to understand what you're actually doing when you create a computer program. Perhaps the most basic description of a computer program is that it is a series of operations that manipulate data. This is true even of the most complicated examples, such as vast, multifeatured Windows applications (e.g., Microsoft Office Suite). Although this is often completely hidden from users of applications, it is always going on behind the scenes.

To illustrate this further, consider the display unit of your computer. What you see onscreen is often so familiar that it is difficult to imagine it as anything other than a "moving picture." In fact, what you see is only a representation of some data, which in its raw form is merely a stream of 0s and 1s stashed away somewhere in the computer's memory. Any onscreen action — moving a mouse pointer, clicking on an icon, typing text into a word processor — results in the shunting around of data in memory.

Of course, simpler situations show this just as well. When using a calculator application, you are supplying data as numbers and performing operations on the numbers in much the same way as you would with paper and pencil — but a lot quicker!

If computer programs are fundamentally performing operations on data, this implies that you need a way to store that data, and some methods to manipulate it. These two functions are provided by *variables* and *expressions*, respectively, and this chapter explores what that means, both in general and specific terms.

First, though, you'll take a look at the basic syntax involved in C# programming, because you need a context in which you can learn about and use variables and expressions in the C# language.

BASIC C# SYNTAX

The look and feel of C# code is similar to that of C++ and Java. This syntax can look quite confusing at first and it's a lot less like written English than some other languages. However, as you immerse yourself in the world of C# programming, you'll find that the style used is a sensible one, and it is possible to write very readable code without much effort.

Unlike the compilers of some other languages, C# compilers ignore additional spacing in code, whether it results from spaces, carriage returns, or tab characters (collectively known as *whitespace characters*). This means you have a lot of freedom in the way that you format your code, although conforming to certain rules can help make things easier to read.

C# code is made up of a series of *statements,* each of which is terminated with a semicolon. Because whitespace is ignored, multiple statements can appear on one line, although for readability it is usual to add carriage returns after semicolons, to avoid multiple statements on one line. It is perfectly acceptable (and quite normal), however, to use statements that span several lines of code.

C# is a *block-structured language,* meaning statements are part of a *block* of code. These blocks, which are delimited with curly brackets (`{` and `}`), may contain any number of statements, or none at all. Note that the curly bracket characters do not need accompanying semicolons.

For example, a simple block of C# code could take the following form:

```
{
   <code line 1, statement 1>;
   <code line 2, statement 2>
      <code line 3, statement 2>;
}
```

Here the `<code line x, statement y>` sections are not actual pieces of C# code; this text is used as a placeholder where C# statements would go. In this case, the second and third lines of code are part of the same statement, because there is no semicolon after the second line.

The following simple example uses *indentation* to clarify the C# itself. This is actually standard practice, and in fact VS automatically does this for you by default. In general, each block of code has its own level of indentation, meaning how far to the right it is. Blocks of code may be *nested* inside each other (that is, blocks may contain other blocks), in which case nested blocks will be indented further:

```
{
   <code line 1>;
   {
      <code line 2>;
      <code line 3>;
   }
   <code line 4>;
}
```

In addition, lines of code that are continuations of previous lines are usually indented further as well, as in the third line of code in the first example above.

NOTE *Look in the VCE or VS Options dialog box (select Tools ➪ Options) to see the rules that VCE uses for formatting your code. There are very many of these, in subcategories of the Text Editor ➪ C# ➪ Formatting node. Most of the settings here reflect parts of C# that haven't been covered yet, but you might want to return to these settings later if you want to tweak them to suit your personal style better. For clarity, this book shows all code snippets as they would be formatted by the default settings.*

Of course, this style is by no means mandatory. If you don't use it, however, you will quickly find that things can get very confusing as you move through this book!

Something else you often see in C# code are *comments*. A comment is not, strictly speaking, C# code at all, but it happily cohabits with it. Comments are self-explanatory: They enable you to add descriptive text to your code — in plain English (or French, German, Mongolian, and so on) — which is ignored by the compiler. When you start dealing with lengthy code sections, it's useful to add reminders about exactly what you are doing, such as "this line of code asks the user for a number" or "this code section was written by Bob."

C# provides two ways of doing this. You can either place markers at the beginning and end of a comment or you can use a marker that means "everything on the rest of this line is a comment." The latter method is an exception to the rule mentioned previously about C# compilers ignoring carriage returns, but it is a special case.

To indicate comments using the first method, you use `/*` characters at the start of the comment and `*/` characters at the end. These may occur on a single line, or on different lines, in which case all lines in between are part of the comment. The only thing you can't type in the body of a comment is `*/`, because that is interpreted as the end marker. For example, the following are OK:

```
/* This is a comment */

/* And so...

                ... is this! */
```

The following, however, causes problems:

```
/* Comments often end with "*/" characters */
```

Here, the end of the comment (the characters after `"*/"`) will be interpreted as C# code, and errors will occur.

The other commenting approach involves starting a comment with `//`. After that, you can write whatever you like — as long as you keep to one line! The following is OK:

```
// This is a different sort of comment.
```

The following fails, however, because the second line is interpreted as C# code:

```
// So is this,
   but this bit isn't.
```

This sort of commenting is useful to document statements because both can be placed on a single line:

```
<A statement>;          // Explanation of statement
```

It was stated earlier that there are two ways of commenting C# code, but there is a third type of comment in C# — although strictly speaking this is an extension of the `//` syntax. You can use single-line comments that start with three `/` symbols instead of two, like this:

```
/// A special comment
```

Under normal circumstances, they are ignored by the compiler — just like other comments — but you can configure VS to extract the text after these comments and create a specially formatted text file when a project is compiled. You can then use it to create documentation. In order for this documentation to be created, the comments must follow the rules of XML documentation — a subject not covered in this book but one that is well worth learning about if you have some spare time.

A *very* important point about C# code is that it is *case sensitive*. Unlike some other languages, you must enter code using exactly the right case, because using an uppercase letter instead of a lowercase one will prevent a project from compiling. For example, consider the following line of code, taken from Chapter 2:

```
Console.WriteLine("The first app in Beginning C# Programming!");
```

This code is understood by the C# compiler, as the case of the `Console.WriteLine()` command is correct. However, none of the following lines of code work:

```
console.WriteLine("The first app in Beginning C# Programming!");
CONSOLE.WRITELINE("The first app in Beginning C# Programming!");
Console.Writeline("The first app in Beginning C# Programming!");
```

Here the case used is wrong, so the C# compiler won't know what you want. Luckily, as you will soon discover, VCE is very helpful when it comes to entering code, and most of the time it knows (as much as a program can know) what you are trying to do. As you type, it suggests commands that you might like to use, and it tries to correct case problems.

BASIC C# CONSOLE APPLICATION STRUCTURE

Let's take a closer look at the console application example from Chapter 2 (ConsoleApplication1), and break down the structure a bit. Here's the code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // Output text to the screen.
         Console.WriteLine("The first app in Beginning C# Programming!");
         Console.ReadKey();
      }
   }
}
```

You can immediately see that all the syntactic elements discussed in the previous section are present here — semicolons, curly braces, and comments, along with appropriate indentation.

The most important section of code at the moment is the following:

```
static void Main(string[] args)
{
   // Output text to the screen.
   Console.WriteLine("The first app in Beginning C# Programming!");
   Console.ReadKey();
}
```

This is the code that is executed when you run your console application. Well, to be more precise, the code block enclosed in curly braces is executed. The comment line doesn't do anything, as mentioned earlier; it's just there for clarity. The other two code lines output some text to the console window and wait for a response, respectively, though the exact mechanisms of this don't concern us for now.

Note how to achieve the code outlining functionality shown in the previous chapter, albeit for a Windows application, since it is such a useful feature. You can do this with the `#region` and `#endregion` keywords, which define the start and end of a region of code that can be expanded and collapsed. For example, you could modify the generated code for ConsoleApplication1 as follows:

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

#endregion
```

This enables you to collapse this code into a single line and expand it again later should you want to look at the details. The `using` statements contained here, and the `namespace` statement just underneath, are explained at the end of this chapter.

NOTE *Any keyword that starts with a `#` is actually a preprocessor directive and not, strictly speaking, a C# keyword. Other than the two described here, `#region` and `#endregion`, these can be quite complicated, and they have very specialized uses. This is one subject you might like to investigate yourself after you have worked through this book.*

For now, don't worry about the other code in the example, because the purpose of these first few chapters is to explain basic C# syntax, so the exact method of how the application execution gets to the point where `Console.WriteLine()` is called is of no concern. Later, the significance of this additional code is made clear.

VARIABLES

As mentioned earlier, variables are concerned with the storage of data. Essentially, you can think of variables in computer memory as boxes sitting on a shelf. You can put things in boxes and take them

out again, or you can just look inside a box to see if anything is there. The same goes for variables; you place data in them and can take it out or look at it, as required.

Although all data in a computer is effectively the same thing (a series of 0s and 1s), variables come in different flavors, known as *types*. Using the box analogy again, boxes come in different shapes and sizes, so some items fit only in certain boxes. The reasoning behind this type system is that different types of data may require different methods of manipulation, and by restricting variables to individual types you can avoid mixing them up. For example, it wouldn't make much sense to treat the series of 0s and 1s that make up a digital picture as an audio file.

To use variables, you have to *declare* them. This means that you have to assign them a *name* and a *type*. After you have declared variables, you can use them as storage units for the type of data that you declared them to hold.

C# syntax for declaring variables merely specifies the type and variable name:

```
<type> <name>;
```

If you try to use a variable that hasn't been declared, your code won't compile, but in this case the compiler tells you exactly what the problem is, so this isn't really a disastrous error. Trying to use a variable without assigning it a value also causes an error, but, again, the compiler detects this.

There are an almost infinite number of types that you can use. This is because you can define your own types to hold whatever convoluted data you like. Having said this, though, there are certain types of data that just about everyone will need to use at some point or another, such as a variable that stores a number. Therefore, you should be aware of several simple, predefined types.

Simple Types

Simple types include types such as numbers and Boolean (true or false) values that make up the fundamental building blocks for your applications. Unlike complex types, simple types cannot have children or attributes. Most of the simple types available are numeric, which at first glance seems a bit strange — surely, you only need one type to store a number?

The reason for the plethora of numeric types is because of the mechanics of storing numbers as a series of 0s and 1s in the memory of a computer. For integer values, you simply take a number of *bits* (individual digits that can be 0 or 1) and represent your number in binary format. A variable storing N bits enables you to represent any number between 0 and $(2^N - 1)$. Any numbers above this value are too big to fit into this variable.

For example, suppose you have a variable that can store two bits. The mapping between integers and the bits representing those integers is therefore as follows:

```
0 = 00
1 = 01
2 = 10
3 = 11
```

In order to store more numbers, you need more bits (three bits enable you to store the numbers from 0 to 7, for example).

The inevitable result of this system is that you would need an infinite number of bits to be able to store every imaginable number, which isn't going to fit in your trusty PC. Even if there were a quantity of

bits you could use for every number, it surely wouldn't be efficient to use all these bits for a variable that, for example, was required to store only the numbers between 0 and 10 (because storage would be wasted). Four bits would do the job fine here, enabling you to store many more values in this range in the same space of memory.

Instead, a number of different integer types can be used to store various ranges of numbers, which take up differing amounts of memory (up to 64 bits). These types are shown in the following table.

NOTE *Each of these types uses one of the standard types defined in the .NET Framework. As discussed in Chapter 1, this use of standard types is what enables language interoperability. The names you use for these types in C# are aliases for the types defined in the framework. The following table lists the names of these types as they are referred to in the .NET Framework library.*

TYPE	ALIAS FOR	ALLOWED VALUES
`sbyte`	`System.SByte`	Integer between –128 and 127
`byte`	`System.Byte`	Integer between 0 and 255
`short`	`System.Int16`	Integer between –32768 and 32767
`ushort`	`System.UInt16`	Integer between 0 and 65535
`int`	`System.Int32`	Integer between –2147483648 and 2147483647
`uint`	`System.UInt32`	Integer between 0 and 4294967295
`long`	`System.Int64`	Integer between –9223372036854775808 and 9223372036854775807
`ulong`	`System.UInt64`	Integer between 0 and 18446744073709551615

The `u` characters before some variable names are shorthand for *unsigned*, meaning that you can't store negative numbers in variables of those types, as shown in the Allowed Values column of the table.

Of course, you also need to store *floating-point* values, those that aren't whole numbers. You can use three floating-point variable types: `float`, `double`, and `decimal`. The first two store floating points in the form $+/-\texttt{m} \times 2^{\texttt{e}}$, where the allowed values for `m` and `e` differ for each type. `decimal` uses the alternative form $+/-\texttt{m} \times 10^{\texttt{e}}$. These three types are shown in the following table, along with their allowed values of `m` and `e`, and these limits in real numeric terms:

TYPE	ALIAS FOR	MIN M	MAX M	MIN E	MAX E	APPROX. MIN VALUE	APPROX. MAX VALUE
`float`	`System.Single`	0	2^{24}	–149	104	1.5×10^{-45}	3.4×10^{38}
`double`	`System.Double`	0	2^{53}	–1075	970	5.0×10^{-324}	1.7×10^{308}
`decimal`	`System.Decimal`	0	2^{96}	–28	0	1.0×10^{-28}	7.9×10^{28}

In addition to numeric types, three other simple types are available:

TYPE	ALIAS FOR	ALLOWED VALUES
`char`	`System.Char`	Single Unicode character, stored as an integer between 0 and 65535
`bool`	`System.Boolean`	Boolean value, `true` or `false`
`string`	`System.String`	A sequence of characters

Note that there is no upper limit on the amount of characters making up a `string`, because it can use varying amounts of memory.

The Boolean type `bool` is one of the most commonly used variable types in C#, and indeed similar types are equally prolific in code in other languages. Having a variable that can be either `true` or `false` has important ramifications when it comes to the flow of logic in an application. As a simple example, consider how many questions can be answered with true or false (or yes and no). Performing comparisons between variable values or validating input are just two of the programmatic uses of Boolean variables that you will examine very soon.

Now that you've seen these types, consider a short example that declares and uses them. In the following Try It Out you use some simple code that declares two variables, assigns them values, and then outputs these values.

TRY IT OUT Using Simple Type Variables

1. Create a new console application called Ch03Ex01 and save it in the directory `C:\BegVCSharp\Chapter03`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   int myInteger;
   string myString;
   myInteger = 17;
   myString = "\"myInteger\" is";
   Console.WriteLine("{0} {1}.", myString, myInteger);
   Console.ReadKey();
}
```

Code snippet Ch03Ex01\Program.cs

3. Execute the code. The result is shown in Figure 3-1.

FIGURE 3-1

How It Works

The added code does three things:

- It declares two variables.
- It assigns values to those two variables.
- It outputs the values of the two variables to the console.

Variable declaration occurs in the following code:

```
int myInteger;
string myString;
```

The first line declares a variable of type `int` with a name of `myInteger`, and the second line declares a variable of type `string` called `myString`.

NOTE *Variable naming is restricted; you can't use just any sequence of characters. You learn about this in the section "Variable Naming."*

The next two lines of code assign values:

```
myInteger = 17;
myString = "\"myInteger\" is";
```

Here you assign two fixed values (known as *literal* values in code) to your variables using the `=` *assignment operator* (the "Expressions" section of this chapter has more details about operators). You assign the integer value `17` to `myInteger`, and the string `"myInteger"` (including the quotes) to `myString`. When you assign string literal values in this way, double quotation marks are required to enclose the string. Therefore, certain characters might cause problems if they are included in the string itself, such as the double quotation characters, and you must escape some characters by substituting a sequence of other characters (an *escape sequence*) that represents the character(s) you want to use. In this example, you use the sequence `\"` to escape a double quotation mark:

```
myString = "\"myInteger\" is";
```

If you didn't use these escape sequences and tried coding this as follows, you would get a compiler error:

```
myString = ""myInteger" is";
```

Note that assigning string literals is another situation in which you must be careful with line breaks — the C# compiler rejects string literals that span more than one line. If you want to add a line break, then use the escape sequence for a new line character in your string, which is `\n`. For example, the assignment

```
myString = "This string has a\nline break.";
```

would be displayed on two lines in the console view as follows:

```
This string has a
line break.
```

All escape sequences consist of the backslash symbol followed by one of a small set of characters (you'll see the full set later). Because this symbol is used for this purpose, there is also an escape sequence for the backslash symbol itself, which is simply two consecutive backslashes (`\\`).

Getting back to the code, there is one more new line to look at:

```
Console.WriteLine("{0} {1}.", myString, myInteger);
```

This looks similar to the simple method of writing text to the console that you saw in the first example, but now you are specifying your variables. To avoid getting ahead of ourselves here, we'll avoid a lot of the details about this line of code at this point. Suffice it to say that it is the technique you will be using in the first part of this book to output text to the console window. Within the brackets you have two things:

- A string
- A list of variables whose values you want to insert into the output string, separated by commas

The string you are outputting, `"{0} {1}."`, doesn't seem to contain much useful text. As shown earlier, however, this is not what you actually see when you run the code. This is because the string is actually a template into which you insert the contents of your variables. Each set of curly brackets in the string is a placeholder that will contain the contents of one of the variables in the list. Each placeholder (or format string) is represented as an integer enclosed in curly brackets. The integers start at 0 and are incremented by 1, and the total number of placeholders should match the number of variables specified in the comma-separated list following the string. When the text is output to the console, each placeholder is replaced by the corresponding value for each variable. In the preceding example, the `{0}` is replaced with the actual value of the first variable, `myString`, and `{1}` is replaced with the contents of `myInteger`.

This method of outputting text to the console is what you use to display output from your code in the examples that follow. Finally, the code includes the line shown in the earlier example for waiting for user input before terminating:

```
Console.ReadKey();
```

Again, the code isn't dissected now, but you will see it frequently in later examples. For now, understand that it pauses code execution until you press a key.

Variable Naming

As mentioned in the previous section, you can't just choose any sequence of characters as a variable name. This isn't as worrying as it might sound, however, because you're still left with a very flexible naming system.

The basic variable naming rules are as follows:

- The first character of a variable name must be either a letter, an underscore character (_), or the *at* symbol (@).
- Subsequent characters may be letters, underscore characters, or numbers.

There are also certain keywords that have a specialized meaning to the C# compiler, such as the `using` and `namespace` keywords shown earlier. If you use one of these by mistake, the compiler complains, however, so don't worry about it.

For example, the following variable names are fine:

```
myBigVar
VAR1
_test
```

These are not, however:

```
99BottlesOfBeer
namespace
It's-All-Over
```

Remember that C# is case sensitive, so be careful not to forget the exact case used when you declare your variables. References to them made later in the program with even so much as a single letter in the wrong case prevents compilation. A further consequence of this is that you can have multiple variables whose names differ only in case. For example, the following are all separate names:

```
myVariable
MyVariable
MYVARIABLE
```

Naming Conventions

Variable names are something you will use *a lot,* so it's worth spending a bit of time learning the sort of names you should use. Before you get started, though, bear in mind that this is controversial ground. Over the years, different systems have come and gone, and many developers will fight tooth and nail to justify their personal system.

Until recently the most popular system was what is known as *Hungarian notation*. This system involves placing a lowercase prefix on all variable names that identify the type. For example, if a variable were of type `int`, then you might place an `i` (or `n`) in front of it, for example `iAge`. Using this system, it is easy to see a variable's type at a glance.

More modern languages, however, such as C#, make this system tricky to implement. For the types you've seen so far, you could probably come up with one- or two-letter prefixes signifying each type. However, because you can create your own types, and there are many hundreds of these more complex types in the basic .NET Framework, this quickly becomes unworkable. With several people working on a project, it would be easy for different people to come up with different and confusing prefixes, with potentially disastrous consequences.

Developers have realized that it is far better to name variables appropriately for their purpose. If any doubt arises, it is easy enough to determine what the type of a variable is. In VS and VCE, you just have to hover the mouse pointer over a variable name and a pop-up box indicates the type soon enough.

Currently, two naming conventions are used in the .NET Framework namespaces: *PascalCase* and *camelCase*. The case used in the names indicates their usage. They both apply to names that comprise multiple words and they both specify that each word in a name should be in lowercase except for its first letter, which should be uppercase. For camelCase terms, there is an additional rule: The first word should start with a lowercase letter.

The following are camelCase variable names:

```
age
firstName
timeOfDeath
```

These are PascalCase:

```
Age
LastName
WinterOfDiscontent
```

For your simple variables, stick to camelCase. Use PascalCase for certain more advanced naming, which is the Microsoft recommendation. Finally, note that many past naming systems involved frequent use of the underscore character, usually as a separator between words in variable names, such as `yet_another_variable`. This usage is now discouraged (which is just as well; it looks ugly!).

Literal Values

The previous Try It Out showed two examples of literal values: `integer` and `string`. The other variable types also have associated literal values, as shown in the following table. Many of these involve *suffixes*, whereby you add a sequence of characters to the end of the literal value to specify the type desired. Some literals have multiple types, determined at compile time by the compiler based on their context (also shown in the following table).

TYPE(S)	CATEGORY	SUFFIX	EXAMPLE/ALLOWED VALUES
`bool`	Boolean	None	`true` or `false`
`int`, `uint`, `long`, `ulong`	Integer	None	`100`
`uint`, `ulong`	Integer	`u` or `U`	`100U`
`long`, `ulong`	Integer	`l` or `L`	`100L`
`ulong`	Integer	`ul`, `uL`, `Ul`, `UL`, `lu`, `lU`, `Lu`, or `LU`	`100UL`
`float`	Real	`f` or `F`	`1.5F`
`double`	Real	None, `d`, or `D`	`1.5`
`decimal`	Real	`m` or `M`	`1.5M`
`char`	Character	None	`'a'`, or escape sequence
`string`	String	None	`"a ... a"`, may include escape sequences

String Literals

Earlier in the chapter, you saw a few of the escape sequences you can use in `string` literals. Here is a full table of these for reference purposes:

ESCAPE SEQUENCE	CHARACTER PRODUCED	UNICODE VALUE OF CHARACTER
`\'`	Single quotation mark	0x0027
`\"`	Double quotation mark	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\a`	Alert (causes a beep)	0x0007
`\b`	Backspace	0x0008
`\f`	Form feed	0x000C
`\n`	New line	0x000A
`\r`	Carriage return	0x000D
`\t`	Horizontal tab	0x0009
`\v`	Vertical tab	0x000B

The Unicode Value column of the preceding table shows the hexadecimal values of the characters as they are found in the Unicode character set. As well as the preceding, you can specify any Unicode character using a Unicode escape sequence. These consist of the standard `\` character followed by a `u` and a four-digit hexadecimal value (for example, the four digits after the `x` in the preceding table).

This means that the following strings are equivalent:

```
"Karli\'s string."
"Karli\u0027s string."
```

Obviously, you have more versatility using Unicode escape sequences.

You can also specify strings *verbatim*. This means that all characters contained between two double quotation marks are included in the string, including end-of-line characters and characters that would otherwise need escaping. The only exception to this is the escape sequence for the double quotation mark character, which must be specified to avoid ending the string. To do this, place the `@` character before the string:

```
@"Verbatim string literal."
```

This string could just as easily be specified in the normal way, but the following requires this method:

```
@"A short list:
item 1
item 2"
```

Verbatim strings are particularly useful in filenames, as these use plenty of backslash characters. Using normal strings, you'd have to use double backslashes all the way along the string:

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

With verbatim string literals you can make this more readable. The following verbatim string is equivalent to the preceding one:

```
@"C:\Temp\MyDir\MyFile.doc"
```

NOTE *As shown later in the book, strings are reference types, unlike the other types you've seen in this chapter, which are value types. One consequence of this is that strings can also be assigned the value* `null`*, which means that the string variable doesn't reference a string (or anything, for that matter).*

Variable Declaration and Assignment

To recap, recall that you declare variables simply using their type and name:

```
int age;
```

You then assign values to variables using the = assignment operator:

```
age = 25;
```

NOTE *Remember that variables must be initialized before you use them. The preceding assignment could be used as an initialization.*

There are a couple of other things you can do here that you are likely to see in C# code. One, you can declare multiple variables of the same type at the same time by separating their names with commas after the type, as follows:

```
int xSize, ySize;
```

Here, `xSize` and `ySize` are both declared as integer types.

The second technique you are likely to see is assigning values to variables when you declare them, which basically means combining two lines of code:

```
int age = 25;
```

You can use both techniques together:

```
int xSize = 4, ySize = 5;
```

Here, both `xSize` and `ySize` are assigned different values. Note that

```
int xSize, ySize = 5;
```

results in only `ySize` being initialized — `xSize` is just declared, and it still needs to be initialized before it's used.

EXPRESSIONS

Now that you've learned how to declare and initialize variables, it's time to look at manipulating them. C# contains a number of *operators* for this purpose. By combining operators with variables and literal values (together referred to as *operands* when used with operators), you can create *expressions,* which are the basic building blocks of computation.

The operators available range from the simple to the highly complex, some of which you might never encounter outside of mathematical applications. The simple ones include all the basic mathematical operations, such as the + operator to add two operands; the complex ones include manipulations of variable content via the binary representation of this content. There are also logical operators specifically for dealing with Boolean values, and assignment operators such as =.

This chapter focuses on the mathematical and assignment operators, leaving the logical ones for the next chapter, where you examine Boolean logic in the context of controlling program flow.

Operators can be roughly classified into three categories:

- **Unary** — Act on single operands
- **Binary** — Act on two operands
- **Ternary** — Act on three operands

Most operators fall into the binary category, with a few unary ones, and a single ternary one called the *conditional operator* (the conditional operator is a logical one and is discussed in Chapter 4). Let's start by looking at the mathematical operators, which span both the unary and binary categories.

Mathematical Operators

There are five simple mathematical operators, two of which (+ and -) have both binary and unary forms. The following table lists each of these operators, along with a short example of its use and the result when it's used with simple numeric types (integer and floating point).

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
+	Binary	`var1 = var2 + var3;`	`var1` is assigned the value that is the sum of `var2` and `var3`.
-	Binary	`var1 = var2 - var3;`	`var1` is assigned the value that is the value of `var3` subtracted from the value of `var2`.
*	Binary	`var1 = var2 * var3;`	`var1` is assigned the value that is the product of `var2` and `var3`.
/	Binary	`var1 = var2 / var3;`	`var1` is assigned the value that is the result of dividing `var2` by `var3`.
%	Binary	`var1 = var2 % var3;`	`var1` is assigned the value that is the remainder when `var2` is divided by `var3`.
+	Unary	`var1 = +var2;`	`var1` is assigned the value of `var2`.
-	Unary	`var1 = -var2;`	`var1` is assigned the value of `var2` multiplied by `-1`.

NOTE *The + (unary) operator is slightly odd, as it has no effect on the result. It doesn't force values to be positive, as you might assume — if* `var2` *is* `-1`*, then* `+var` *is also* `-1`*. However, it is a universally recognized operator, and as such is included. The most useful fact about this operator is shown later in this book when you look at operator overloading.*

The examples use simple numeric types because the result can be unclear when using the other simple types. What would you expect if you added two Boolean values together, for example? In this case, nothing, because the compiler complains if you try to use + (or any of the other mathematical operators) with `bool` variables. Adding `char` variables is also slightly confusing. Remember that `char` variables are actually stored as numbers, so adding two `char` variables also results in a number (of type `int`, to be precise). This is an example of *implicit conversion,* which you'll learn a lot more about shortly (along with explicit conversion), because it also applies to cases where `var1`, `var2`, and `var3` are of mixed types.

The binary + operator *does* make sense when used with string type variables. In this case, the table entry should read as shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`+`	Binary	`var1 = var2 + var3;`	`var1` is assigned the value that is the concatenation of the two strings stored in `var2` and `var3`.

None of the other mathematical operators, however, work with strings.

The other two operators you should look at here are the increment and decrement operators, both of which are unary operators that can be used in two ways: either immediately before or immediately after the operand. The results obtained in simple expressions are shown in the next table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`++`	Unary	`var1 = ++var2;`	`var1` is assigned the value of `var2 + 1`. `var2` is incremented by `1`.
`--`	Unary	`var1 = --var2;`	`var1` is assigned the value of `var2 - 1`. `var2` is decremented by `1`.
`++`	Unary	`var1 = var2++;`	`var1` is assigned the value of `var2`. `var2` is incremented by `1`.
`--`	Unary	`var1 = var2--;`	`var1` is assigned the value of `var2`. `var2` is decremented by `1`.

These operators always result in a change to the value stored in their operand:

- ++ always results in its operand being incremented by one.
- –– always results in its operand being decremented by one.

The differences between the results stored in `var1` are a consequence of the fact that the placement of the operator determines when it takes effect. Placing one of these operators before its operand means that the operand is affected before any other computation takes place. Placing it after the operand means that the operand is affected after all other computation of the expression is completed.

This merits another example! Consider this code:

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

What value will be assigned to `var1`? Before the expression is evaluated, the `--` operator preceding `var3` takes effect, changing its value from `6` to `5`. You can ignore the `++` operator that follows `var2`, as it won't take effect until after the calculation is completed, so `var1` will be the product of `5` and `5`, or `25`.

These simple unary operators come in very handy in a surprising number of situations. They are really just a shorthand for expressions such as this:

```
var1 = var1 + 1;
```

This sort of expression has many uses, particularly where *looping* is concerned, as shown in the next chapter. The following Try It Out provides an example demonstrating how to use the mathematical operators, and it introduces a couple of other useful concepts as well. The code prompts you to type in a string and two numbers and then demonstrates the results of performing some calculations.

TRY IT OUT Manipulating Variables with Mathematical Operators

1. Create a new console application called Ch03Ex02 and save it to the directory `C:\BegVCSharp\Chapter03`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   double firstNumber, secondNumber;
   string userName;
   Console.WriteLine("Enter your name:");
   userName = Console.ReadLine();
   Console.WriteLine("Welcome {0}!", userName);
   Console.WriteLine("Now give me a number:");
   firstNumber = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("Now give me another number:");
   secondNumber = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
               secondNumber, firstNumber + secondNumber);
   Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
               secondNumber, firstNumber, firstNumber - secondNumber);
```

```
            Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
                              secondNumber, firstNumber * secondNumber);
            Console.WriteLine("The result of dividing {0} by {1} is {2}.",
                              firstNumber, secondNumber, firstNumber / secondNumber);
            Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
                              firstNumber, secondNumber, firstNumber % secondNumber);
            Console.ReadKey();
        }
```

Code snippet Ch03Ex02\Program.cs

3. Execute the code. The display shown in Figure 3-2 appears.

FIGURE 3-2

4. Enter your name and press Enter. Figure 3-3 shows the display.

FIGURE 3-3

5. Enter a number, press Enter, enter another number, and then press Enter again. Figure 3-4 shows an example result.

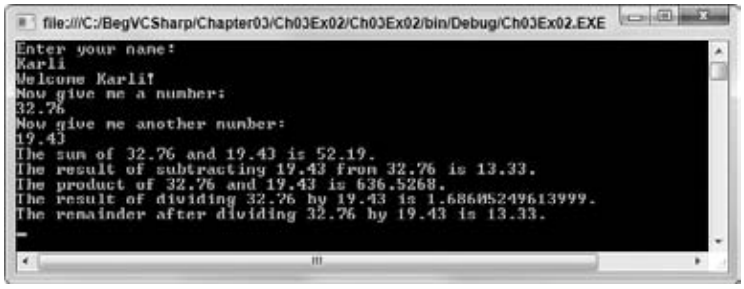

FIGURE 3-4

How It Works

As well as demonstrating the mathematical operators, this code introduces two important concepts that you will often come across:

- User input
- Type conversion

User input uses a syntax similar to the `Console.WriteLine()` command you've already seen — you use `Console.ReadLine()`. This command prompts the user for input, which is stored in a `string` variable:

```
string userName;
Console.WriteLine("Enter your name:");
userName = Console.ReadLine();
Console.WriteLine("Welcome {0}!", userName);
```

This code writes the contents of the assigned variable, `userName`, straight to the screen.

You also read in two numbers in this example. This is slightly more involved, because the `Console.ReadLine()` command generates a string, but you want a number. This introduces the topic of *type conversion*, which is covered in more detail in Chapter 5, but let's have a look at the code used in this example.

First, you declare the variables in which you want to store the number input:

```
double firstNumber, secondNumber;
```

Next, you supply a prompt and use the command `Convert.ToDouble()` on a string obtained by `Console.ReadLine()` to convert the string into a `double` type. You assign this number to the `firstNumber` variable you have declared:

```
Console.WriteLine("Now give me a number:");
firstNumber = Convert.ToDouble(Console.ReadLine());
```

This syntax is remarkably simple, and many other conversions can be performed in a similar way.

The remainder of the code obtains a second number in the same way:

```
Console.WriteLine("Now give me another number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

Next, you output the results of adding, subtracting, multiplying, and dividing the two numbers, in addition to displaying the remainder after division, using the remainder (`%`) operator:

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
            secondNumber, firstNumber + secondNumber);
Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
            secondNumber, firstNumber, firstNumber - secondNumber);
Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
            secondNumber, firstNumber * secondNumber);
Console.WriteLine("The result of dividing {0} by {1} is {2}.",
            firstNumber, secondNumber, firstNumber / secondNumber);
Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
            firstNumber, secondNumber, firstNumber % secondNumber);
```

Note that you are supplying the expressions, `firstNumber + secondNumber` and so on, as a parameter to the `Console.WriteLine()` statement, without using an intermediate variable:

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
                  secondNumber, firstNumber + secondNumber);
```

This kind of syntax can make your code very readable, and reduce the number of lines of code you need to write.

Assignment Operators

So far, you've been using the simple = assignment operator, and it may come as a surprise that any other assignment operators exist at all. There are more, however, and they're quite useful! All of the assignment operators other than = work in a similar way. Like =, they all result in a value being assigned to the variable on their left side based on the operands and operators on their right side.

The following table describes the operators:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
=	Binary	`var1 = var2;`	`var1` is assigned the value of `var2`.
+=	Binary	`var1 += var2;`	`var1` is assigned the value that is the sum of `var1` and `var2`.
-=	Binary	`var1 -= var2;`	`var1` is assigned the value that is the value of `var2` subtracted from the value of `var1`.
*=	Binary	`var1 *= var2;`	`var1` is assigned the value that is the product of `var1` and `var2`.
/=	Binary	`var1 /= var2;`	`var1` is assigned the value that is the result of dividing `var1` by `var2`.
%=	Binary	`var1 %= var2;`	`var1` is assigned the value that is the remainder when `var1` is divided by `var2`.

As you can see, the additional operators result in `var1` being included in the calculation, so code like

```
var1 += var2;
```

has exactly the same result as

```
var1 = var1 + var2;
```

NOTE *The += operator can also be used with strings, just like +.*

Using these operators, especially when employing long variable names, can make code much easier to read.

Operator Precedence

When an expression is evaluated, each operator is processed in sequence, but this doesn't necessarily mean evaluating these operators from left to right. As a trivial example, consider the following:

```
var1 = var2 + var3;
```

Here, the + operator acts before the = operator. There are other situations where operator precedence isn't so obvious, as shown here:

```
var1 = var2 + var3 * var4;
```

In the preceding example, the * operator acts first, followed by the + operator, and finally the = operator. This is standard mathematical order, and it provides the same result as you would expect from working out the equivalent algebraic calculation on paper.

Similarly, you can gain control over operator precedence by using parentheses, as shown in this example:

Here, the content of the parentheses is evaluated first, meaning that the + operator acts before the * operator.

The following table shows the order of precedence for the operators you've encountered so far, whereby operators of equal precedence (such as * and /) are evaluated from left to right:

PRECEDENCE	OPERATORS
Highest	++, -- (used as prefixes); +, - (unary)
	*, /, %
	+, -
	=, *=, /=, %=, +=, -=
Lowest	++, -- (used as suffixes)

NOTE *You can use parentheses to override this precedence order, as described previously. In addition, note that ++ and --, when used as suffixes, only have lowest priority in conceptual terms, as described in the table. They don't operate on the result of, say, an assignment expression, so you can consider them to have a higher priority than all other operators. However, because they change the value of their operand after expression evaluation, it's easier to think of their precedence as shown in the preceding table.*

Namespaces

Before moving on, it's worthwhile to consider one more important subject — *namespaces*. These are the .NET way of providing containers for application code, such that code and its contents may be uniquely

identified. Namespaces are also used as a means of categorizing items in the .NET Framework. Most of these items are type definitions, such as the simple types in this chapter (`System.Int32` and so on).

C# code, by default, is contained in the *global namespace*. This means that items contained in this code are accessible from other code in the global namespace simply by referring to them by name. You can use the `namespace` keyword, however, to explicitly define the namespace for a block of code enclosed in curly brackets. Names in such a namespace must be *qualified* if they are used from code outside of this namespace.

A qualified name is one that contains all of its hierarchical information, which basically means that if you have code in one namespace that needs to use a name defined in a different namespace, you must include a reference to this namespace. Qualified names use period characters (`.`) between namespace levels, as shown here:

```
namespace LevelOne
{
   // code in LevelOne namespace

   // name "NameOne" defined
}

// code in global namespace
```

This code defines one namespace, `LevelOne`, and a name in this namespace, `NameOne` (no actual code is shown here to keep the discussion general; instead, a comment appears where the definition would go). Code written inside the `LevelOne` namespace can simply refer to this name using `NameOne` — no classification is necessary. Code in the global namespace, however, must refer to this name using the classified name `LevelOne.NameOne`.

NOTE *By convention, namespaces are usually written in PascalCase.*

Within a namespace, you can define nested namespaces, also using the `namespace` keyword. Nested namespaces are referred to via their hierarchy, again using periods to classify each level of the hierarchy. This is best illustrated with an example. Consider the following namespaces:

```
namespace LevelOne
{
   // code in LevelOne namespace

   namespace LevelTwo
   {
      // code in LevelOne.LevelTwo namespace

      // name "NameTwo" defined
   }
}

// code in global namespace
```

Here, `NameTwo` must be referred to as `LevelOne.LevelTwo.NameTwo` from the global namespace, `LevelTwo.NameTwo` from the `LevelOne` namespace, and `NameTwo` from the `LevelOne.LevelTwo` namespace.

The important point here is that names are uniquely defined by their namespace. You could define the name `NameThree` in the `LevelOne` and `LevelTwo` namespaces:

```
namespace LevelOne
{
   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

This defines two separate names, `LevelOne.NameThree` and `LevelOne.LevelTwo.NameThree`, which can be used independently of each other.

After namespaces are set up, you can use the `using` statement to simplify access to the names they contain. In effect, the `using` statement says, "OK, I'll be needing names from this namespace, so don't bother asking me to classify them every time." For example, the following code says that code in the `LevelOne` namespace should have access to names in the `LevelOne.LevelTwo` namespace without classification:

```
namespace LevelOne
{
   using LevelTwo;

   namespace LevelTwo
   {
      // name "NameTwo" defined
   }
}
```

Code in the `LevelOne` namespace can now refer to `LevelTwo.NameTwo` by simply using `NameTwo`.

Sometimes, as with the `NameThree` example shown previously, this can lead to problems with clashes between identical names in different namespaces (if you use such a name, then your code won't compile — and the compiler will let you know that there is an ambiguity). In cases such as these, you can provide an *alias* for a namespace as part of the `using` statement:

```
namespace LevelOne
{
   using LT = LevelTwo;

   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

Here, code in the `LevelOne` namespace can refer to `LevelOne.NameThree` as `NameThree` and `LevelOne.LevelTwo.NameThree` as `LT.NameThree`.

`using` statements apply to the namespace they are contained in, and any nested namespaces that might also be contained in this namespace. In the preceding code, the global namespace can't use `LT.NameThree`. However, if this `using` statement were declared as

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

then code in the global namespace and the `LevelOne` namespace could use `LT.NameThree`.

Note one more important point here: The `using` statement doesn't in itself give you access to names in another namespace. Unless the code in a namespace is in some way linked to your project, by being defined in a source file in the project or being defined in some other code linked to the project, you won't have access to the names contained. In addition, if code containing a namespace is linked to your project, then you have access to the names contained in that code, regardless of whether you use `using`. `using` simply makes it easier for you to access these names, and it can shorten otherwise lengthy code to make it more readable.

Going back to the code in `ConsoleApplication1` shown at the beginning of this chapter, the following lines that apply to namespaces appear:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   ...
}
```

The four lines that start with the `using` keyword are used to declare that the `System`, `System.Collections.Generic`, `System.Linq`, and `System.Text` namespaces will be used in this C# code and should be accessible from all namespaces in this file without classification. The `System` namespace is the root namespace for .NET Framework applications and contains all the basic functionality you need for console applications. The other two namespaces are very often used in console applications, so they are there just in case.

Finally, a namespace is declared for the application code itself, `ConsoleApplication1`.

SUMMARY

In this chapter, you covered a fair amount of ground on the way to creating usable (if basic) C# applications. You've looked at the basic C# syntax and analyzed the basic console application code that VS and VCE generate for you when you create a console application project.

The major part of this chapter concerned the use of variables. You have seen what variables are, how you create them, how you assign values to them, and how you manipulate them and the values that they contain. Along the way, you've also looked at some basic user interaction, which showed how you can output text to a console application and read user input back in. This involved some very basic type conversion, a complex subject that is covered in more depth in Chapter 5.

You also learned how you can assemble operators and operands into expressions, and looked at the way these are executed and the order in which this takes place.

Finally, you looked at namespaces, which will become increasingly important as the book progresses. By introducing this topic in a fairly abstract way here, the groundwork is completed for later discussions.

So far, all of your programming has taken the form of line-by-line execution. In the next chapter, you learn how to make your code more efficient by controlling the flow of execution using looping techniques and conditional branching.

EXERCISES

1. In the following code, how would you refer to the name `great` from code in the namespace `fabulous`?

```
namespace fabulous
{
   // code in fabulous namespace
}

namespace super
{
   namespace smashing
   {
      // great name defined
   }
}
```

2. Which of the following is not a legal variable name?

- `myVariableIsGood`
- `99Flake`
- `_floor`
- `time2GetJiggyWidIt`
- `wrox.com`

continues

3. Is the string `"supercalifragilisticexpialidocious"` too big to fit in a `string` variable? If so, why?

4. By considering operator precedence, list the steps involved in the computation of the following expression:

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

5. Write a console application that obtains four `int` values from the user and displays the product. Hint: You may recall that the `Convert.ToDouble()` command was used to convert the input from the console to a `double`; the equivalent command to convert from a `string` to an `int` is `Convert.ToInt32()`.

Answers to Exercises can be found in Appendix A.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Basic C# syntax	C# is a case sensitive language, and each line of code is terminated with a semicolon. Lines can be indented for ease of reading if they get too long, or to identify nested blocks. You can include non-compiled comments with `//` or `/* ... */` syntax. Blocks of code can be collapsed into regions, also to ease readability.
Variables	Variables are chunks of data that have a name and a type. The .NET Framework defines plenty of simple types, such as numeric and string (text) types for you to use. Variables must be declared and initialized for you to use them. You can assign literal values to variables to initialize them, and variables can be declared and initialized in a single step.
Expressions	Expressions are built from operators and operands, where operators perform operations on operands. There are three types of operators — unary, binary, and ternary — that operate on 1, 2, and 3 operands respectively. Mathematical operators perform operations on numeric values, and assignment operators place the result of an expression into a variable. Operators have a fixed precedence that determines the order in which they are processed in an expression.
Namespaces	All names defined in a .NET application, including variable names, are contained in a namespace. Namespaces are hierarchical, and you often have to qualify names according to the namespace that contains them in order to access them.

4

Flow Control

WHAT YOU WILL LEARN IN THIS CHAPTER

- Boolean logic and how to use it
- How to branch code
- How to loop code

All of the C# code you've seen so far has had one thing in common. In each case, program execution has proceeded from one line to the next in top-to-bottom order, missing nothing. If all applications worked like this, then you would be very limited in what you could do. This chapter describes two methods for controlling program flow — that is, the order of execution of lines of C# code: *branching* and *looping*. Branching executes code conditionally, depending on the outcome of an evaluation, such as "only execute this code if the variable `myVal` is less than 10." Looping repeatedly executes the same statements, either a certain number of times or until a test condition has been reached.

Both of these techniques involve the use of *Boolean logic*. In the last chapter you saw the `bool` type, but didn't actually do much with it. This chapter uses it a lot, so the chapter begins by discussing what is meant by Boolean logic so that you can use it in flow control scenarios.

BOOLEAN LOGIC

The `bool` type introduced in the previous chapter can hold one of only two values: `true` or `false`. This type is often used to record the result of some operation, so that you can act on this result. In particular, `bool` types are used to store the result of a *comparison*.

NOTE *As a historical aside, it is the work of the mid-nineteenth-century English mathematician George Boole that forms the basis of Boolean logic.*

For instance, consider the situation (mentioned in the chapter introduction) in which you want to execute code based on whether a variable, `myVal`, is less than 10. To do this, you need some indication of whether the statement "`myVal` is less than 10" is true or false — that is, you need to know the Boolean result of a comparison.

Boolean comparisons require the use of Boolean *comparison operators* (also known as *relational operators*), which are shown in the following table. In all cases here, `var1` is a `bool` type variable, whereas the types of `var2` and `var3` may vary.

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`==`	Binary	`var1 = var2 == var3;`	`var1` is assigned the value `true` if `var2` is equal to `var3`, or `false` otherwise.
`!=`	Binary	`var1 = var2 != var3;`	`var1` is assigned the value `true` if `var2` is not equal to `var3`, or `false` otherwise.
`<`	Binary	`var1 = var2 < var3;`	`var1` is assigned the value `true` if `var2` is less than `var3`, or `false` otherwise.
`>`	Binary	`var1 = var2 > var3;`	`var1` is assigned the value `true` if `var2` is greater than `var3`, or `false` otherwise.
`<=`	Binary	`var1 = var2 <= var3;`	`var1` is assigned the value `true` if `var2` is less than or equal to `var3`, or `false` otherwise.
`>=`	Binary	`var1 = var2 >= var3;`	`var1` is assigned the value `true` if `var2` is greater than or equal to `var3`, or `false` otherwise.

You might use operators such as these on numeric values in code:

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

This code results in `isLessThan10` being assigned the value `true` if `myVal` stores a value less than 10, or `false` otherwise.

You can also use these comparison operators on other types, such as strings:

```
bool isKarli;
isKarli = myString == "Karli";
```

Here, `isKarli` is `true` only if `myString` stores the string `"Karli"`.

You can also compare variables with Boolean values:

```
bool isTrue;
isTrue = myBool == true;
```

Here, however, you are limited to the use of the `==` and `!=` operators.

NOTE *A common code error occurs if you unintentionally assume that because* `val1 < val2` *is false,* `val1 > val2` *is true. If* `val1 == val2`*, then both these statements are false.*

Some other Boolean operators are intended specifically for working with Boolean values, as shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`!`	Unary	`var1 = !var2;`	`var1` is assigned the value `true` if `var2` is `false`, or `false` if `var2` is `true`. (Logical `NOT`)
`&`	Binary	`var1 = var2 & var3;`	`var1` is assigned the value `true` if `var2` and `var3` are both `true`, or `false` otherwise. (Logical `AND`)
`|`	Binary	`var1 = var2 | var3;`	`var1` is assigned the value `true` if either `var2` or `var3` (or both) is `true`, or `false` otherwise. (Logical `OR`)
`^`	Binary	`var1 = var2 ^ var3;`	`var1` is assigned the value `true` if either `var2` or `var3`, but not both, is `true`, or `false` otherwise. (Logical `XOR` or exclusive `OR`)

Therefore, the previous code snippet could also be expressed as follows:

```
bool isTrue;
isTrue = myBool & true;
```

The `&` and `|` operators also have two similar operators, known as *conditional Boolean* operators, shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`&&`	Binary	`var1 = var2 && var3;`	`var1` is assigned the value `true` if `var2` and `var3` are both `true`, or `false` otherwise. (Logical `AND`)
`||`	Binary	`var1 = var2 || var3;`	`var1` is assigned the value `true` if either `var2` or `var3` (or both) is `true`, or `false` otherwise. (Logical `OR`)

The result of these operators is exactly the same as `&` and `|`, but there is an important difference in the way this result is obtained, which can result in better performance. Both of these look at the value of their first operands (`var2` in the preceding table) and, based on the value of this operand, may not need to process the second operands (`var3` in the preceding table) at all.

If the value of the first operand of the `&&` operator is `false`, then there is no need to consider the value of the second operand, because the result will be `false` regardless. Similarly, the `||` operator returns `true` if its first operand is `true`, regardless of the value of the second operand. This isn't the case for the `&` and `|` operators shown earlier. With these, both operands are always evaluated.

Because of this conditional evaluation of operands, you get a small performance increase if you use `&&` and `||` instead of `&` and `|`. This is particularly apparent in applications that use these operators a lot. As a rule of thumb, *always* use `&&` and `||` where possible. These operators really come into their own in more complicated situations, where computation of the second operand is possible only with certain values of the first operand, as shown in this example:

```
var1 = (var2 != 0) && (var3 / var2 > 2);
```

Here, if `var2` is zero, then dividing `var3` by `var2` results in either a "division by zero" error or `var1` being defined as infinite (the latter is possible, and detectable, with some types, such as `float`).

NOTE *At this point, you may be asking why the `&` and `|` operators exist at all. The reason is that these operators may be used to perform operations on numeric values. In fact, as you will see shortly in the section "Bitwise Operators," they operate on the series of bits stored in a variable, rather than the value of the variable.*

Boolean Assignment Operators

Boolean comparisons can be combined with assignments by using Boolean assignment operators. These work in the same way as the mathematical assignment operators that were introduced in the preceding chapter (`+=`, `*=`, and so on). The Boolean versions are shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`&=`	Binary	`var1 &= var2;`	`var1` is assigned the value that is the result of `var1 & var2`.
`|=`	Binary	`var1 |= var2;`	`var1` is assigned the value that is the result of `var1 | var2`.
`^=`	Binary	`var1 ^= var2;`	`var1` is assigned the value that is the result of `var1 ^ var2`.

These work with both Boolean and numeric values in the same way as `&`, `|`, and `^`.

NOTE *Note that the `&=` and `|=` assignment operators do not make use of the `&&` and `||` conditional Boolean operators; that is, all operands are processed regardless of the value to the left of the assignment operator.*

In the Try It Out that follows, you type in an integer and then the code performs various Boolean evaluations using that integer.

TRY IT OUT Using Boolean Operators

1. Create a new console application called Ch04Ex01 and save it in the directory `C:\BegVCSharp\Chapter04`.

2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   Console.WriteLine("Enter an integer:");
   int myInt = Convert.ToInt32(Console.ReadLine());
   bool isLessThan10 = myInt < 10;
   bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
   Console.WriteLine("Integer less than 10? {0}", isLessThan10);
   Console.WriteLine("Integer between 0 and 5? {0}", isBetween0And5);
   Console.WriteLine("Exactly one of the above is true? {0}",
       isLessThan10 ^ isBetween0And5);
   Console.ReadKey();
}
```

Code snippet Ch04Ex01\Program.cs

3. Execute the application and enter an integer when prompted. The result is shown in Figure 4-1.

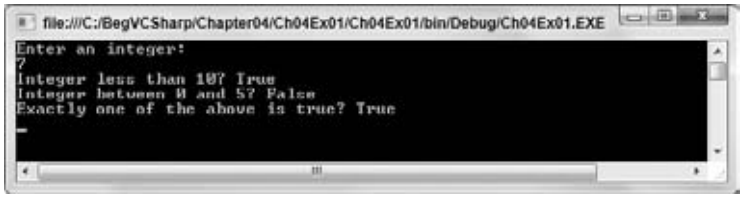

FIGURE 4-1

How It Works

The first two lines of code prompt for and accept an integer value using techniques you've already seen:

```
Console.WriteLine("Enter an integer:");
int myInt = Convert.ToInt32(Console.ReadLine());
```

You use `Convert.ToInt32()` to obtain an integer from the string input, which is simply another conversion command in the same family as the `Convert.ToDouble()` command used previously.

Next, two Boolean variables, `isLessThan10` and `isBetween0And5`, are declared and assigned values with logic that matches the description in their names:

```
bool isLessThan10 = myInt < 10;
bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
```

These variables are used in the next three lines of code, the first two of which output their values, while the third performs an operation on them and outputs the result. You work through this code assuming that the user enters 7, as shown in the screenshot.

The first output is the result of the operation `myInt < 10`. If `myInt` is 6, which is less than 10, then the result is `true`, which is what you see displayed. Values of `myInt` of 10 or higher result in `false`.

The second output is a more involved calculation: `(0 <= myInt) && (myInt <= 5)`. This involves two comparison operations, to determine whether `myInt` is greater than or equal to 0 and less than or equal to 5, and a Boolean `AND` operation on the results obtained. With a value of 6, `(0 <= myInt)` returns `true`, and `(myInt <= 5)` returns `false`. The result is then `(true) && (false)`, which is `false`, as you can see from the display.

Finally, you perform a logical exclusive `OR` on the two Boolean variables `isLessThan10` and `isBetween0And5`. This will return `true` if one of the values is `true` and the other `false`, so only if `myInt` is 6, 7, 8, or 9. With a value of 6, as in the example, the result is `true`.

Bitwise Operators

The `&` and `|` operators you saw earlier serve an additional purpose: They may be used to perform operations on numeric values. When used in this way, they operate on the series of bits stored in a variable, rather than the value of the variable, which is why they are referred to as *bitwise* operators.

In this section you will look at these and other bitwise operators that are defined by the C# language. Using this functionality is fairly uncommon in most development, apart from mathematical applications. For that reason there is no Try it Out for this section.

Let's start by considering `&` and `|` in turn. Each bit in the first operand is compared with the bit in the same position in the second operand, resulting in the bit in the same position in the resultant value being assigned a value, as shown here:

OPERAND 1 BIT	OPERAND 2 BIT	& RESULT BIT
1	1	1
1	0	0
0	1	0
0	0	0

`|` is similar, but the result bits are different:

OPERAND 1 BIT	OPERAND 2 BIT	| RESULT BIT
1	1	1
1	0	1
0	1	1
0	0	0

For example, consider the operation shown here:

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = op1 & op2;
```

In this case, you must consider the binary representations of `op1` and `op2`, which are 100 and 101, respectively. The result is obtained by comparing the binary digits in equivalent positions in these two representations as follows:

- The leftmost bit of `result` is 1 if the leftmost bits of `op1` and `op2` are both 1, or 0 otherwise.
- The next bit of `result` is 1 if the next bits of `op1` and `op2` are both 1, or 0 otherwise.
- Continue for all remaining bits.

In this example, the leftmost bits of `op1` and `op2` are both 1, so the leftmost bit of `result` will be 1, too. The next bits are both 0, and the third bits are 1 and 0, respectively, so the second and third bits of `result` will be 0. The final value of `result` in binary representation is therefore 100, so the result is assigned the value 4. This is shown graphically in the following equations:

$$\begin{array}{rrrr} & 1 & 0 & 0 \\ \& & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 \end{array} \qquad \begin{array}{rr} & 4 \\ \& & 5 \\ \hline & 4 \end{array}$$

The same process occurs if you use the | operator, except that in this case each result bit is 1 if either of the operand bits in the same position is 1, as shown in the following equations:

$$\begin{array}{rrrr} & 1 & 0 & 0 \\ | & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 \end{array} \qquad \begin{array}{rr} & 4 \\ | & 5 \\ \hline & 5 \end{array}$$

You can also use the ^ operator in the same way, where each result bit is 1 if one or other of the operand bits in the same position is 1, but not both, as shown in the following table:

OPERAND 1 BIT	OPERAND 2 BIT	^ RESULT BIT
1	1	0
1	0	1
0	1	1
0	0	0

C# also allows the use of a unary bitwise operator (~), which acts on its operand by inverting each of its bits, so that the result is a variable having values of 1 for each bit in the operand that is 0, and vice versa. This is shown in the following table:

OPERAND BIT	~ RESULT BIT
1	0
0	1

The way integer numbers are stored in .NET, known as *two's complement*, means that using the ~ unary operator can lead to results that look a little odd. If you remember that an `int` type is a 32-bit number, for example, then knowing that the ~ operator acts on all 32 of those bits can

help you to see what is going on. For example, the number 5 in its full binary representation is as follows:

```
00000000000000000000000000000101
```

This is the number –5:

```
11111111111111111111111111111011
```

In fact, by the two's complement system, (–x) is defined as (~x + 1). That may seem odd, but this system is very useful when it comes to adding numbers. For example, adding 10 and –5 (that is, subtracting 5 from 10) looks like this in binary format:

```
   00000000000000000000000000001010
+  11111111111111111111111111111011
= 100000000000000000000000000000101
```

NOTE *By ignoring the 1 on the far left, you are left with the binary representation for 5, so while results such as ~1 = –2 may look odd, the underlying structures force this result.*

The bitwise operations you've seen in this section are quite useful in certain situations, because they enable an easy method of using individual variable bits to store information. Consider a simple representation of a color using three bits to specify `red`, `green`, and `blue` content. You can set these bits independently to change the three bits to one of the configurations shown in the following table:

BITS	DECIMAL REPRESENTATION	MEANING
000	0	black
100	4	red
010	2	green
001	1	blue
101	5	magenta
110	6	yellow
011	3	cyan
111	7	white

Suppose you store these values in a variable of type `int`. Starting from a black color — that is, an `int` variable with the value of 0 — you can perform operations like this:

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;            // Add green bit, myColor now stores 010
myColor = myColor | 4;            // Add red bit, myColor now stores 110
containsRed = (myColor & 4) == 4; // Check value of red bit
```

The final line of code assigns a value of `true` to `containsRed`, as the red bit of `myColor` is 1. This technique can be quite useful for making efficient use of information, particularly because the operations involved can be used to check the values of multiple bits simultaneously (32 in the case of `int` values). However, there are better ways to store extra information in single variables (making use of the advanced variable types discussed in the next chapter).

In addition to these four bitwise operators, this section considers two others, shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`>>`	Binary	`var1 = var2 >> var3;`	`var1` is assigned the value obtained when the binary content of `var2` is shifted `var3` bits to the right.
`<<`	Binary	`var1 = var2 << var3;`	`var1` is assigned the value obtained when the binary content of `var2` is shifted `var3` bits to the left.

These operators, commonly called *bitwise shift operators,* are best illustrated with a quick example:

```
int var1, var2 = 10, var3 = 2;
var1 = var2 << var3;
```

Here, `var1` is assigned the value 40. This can be explained by considering that the binary representation of 10 is 1010, which shifted to the left by two places is 101000 — the binary representation of 40. In effect, you have carried out a multiplication operation. Each bit shifted to the left multiplies the value by 2, so two bit-shifts to the left result in multiplication by 4. Conversely, each bit shifted to the right has the effect of dividing the operand by 2, with any non-integer remainder being lost:

```
int var1, var2 = 10;
var1 = var2 >> 1;
```

In this example, `var1` contains the value 5, whereas the following code results in a value of 2:

```
int var1, var2 = 10;
var1 = var2 >> 2;
```

You are unlikely to use these operators in most code, but it is worth being aware of their existence. Their primary use is in highly optimized code, where the overhead of other mathematical operations just won't do. For this reason, they are often used in, for example, device drivers or system code.

The bitwise shift operators also have assignment operators, as shown in the following table:

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
`>>=`	Unary	`var1 >>= var2;`	`var1` is assigned the value obtained when the binary content of `var1` is shifted `var2` bits to the right.
`<<=`	Unary	`var1 <<= var2;`	`var1` is assigned the value obtained when the binary content of `var1` is shifted `var2` bits to the left.

Operator Precedence Updated

Now that you have a few more operators to consider, the operator precedence table from the previous chapter should be updated to include them. The new order is shown in the following table:

PRECEDENCE	OPERATORS
Highest	++, -- (used as prefixes); (), +, - (unary), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	|
	&&
	||
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
Lowest	++, -- (used as suffixes)

This adds quite a few more levels but explicitly defines how expressions such as the following will be evaluated, where the `&&` operator is processed after the `<=` and `>=` operators (in this code `var2` is an `int` value):

```
var1 = var2 <= 4 && var2 >= 2;
```

It doesn't hurt to add parentheses to make expressions such as this one clearer. The compiler knows what order to process operators in, but we humans are prone to forget such things (and you might want to change the order). Writing the previous expression as

```
var1 = (var2 <= 4) && (var2 >= 2);
```

solves this problem by explicitly ordering the computation.

THE GOTO STATEMENT

C# enables you to label lines of code and then jump straight to them using the `goto` statement. This has its benefits and problems. The main benefit is that it's a simple way to control what code is executed when. The main problem is that excessive use of this technique can result in spaghetti code that is difficult to understand.

The `goto` statement is used as follows:

```
goto <labelName>;
```

Labels are defined as follows:

```
<labelName>:
```

For example, consider the following:

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

Execution proceeds as follows:

- `myInteger` is declared as an `int` type and assigned the value `5`.
- The `goto` statement interrupts normal execution and transfers control to the line marked `myLabel:`.
- The value of `myInteger` is written to the console.

The highlighted line in the following code is never executed:

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

In fact, if you try to compile this code in an application, the Error List window will show a warning labeled "Unreachable code detected," along with location details. You will also see a wavy green line under `myInteger` on the unreachable line of code.

`goto` statements have their uses, but they can make things very confusing indeed. In fact, if you can avoid it (and by using the techniques you'll learn in the remainder of this chapter you'll be able to), never use `goto`. The following example shows some spaghetti code arising from the use of this unfortunate keyword:

```
start:
int myInteger = 5;
goto addVal;
writeResult:
Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal:
myInteger += 10;
goto writeResult;
```

This is perfectly valid code but very difficult to read! Try it out for yourself and see what happens. Before doing that, though, try to first determine what this code will do by looking at it, and then give yourself a pat on the back if you're right. You'll revisit this statement a little later, because it has implications for use with some of the other structures in this chapter.

BRANCHING

Branching is the act of controlling which line of code should be executed next. The line to jump to is controlled by some kind of conditional statement. This conditional statement is based on a comparison between a test value and one or more possible values using Boolean logic.

This section describes three branching techniques available in C#:

- The ternary operator
- The `if` statement
- The `switch` statement

The Ternary Operator

The simplest way to perform a comparison is to use the *ternary* (or *conditional*) operator mentioned in the last chapter. You've already seen unary operators that work on one operand, and binary operators that work on two operands, so it won't come as a surprise that this operator works on three operands. The syntax is as follows:

```
<test> ? <resultIfTrue>: <resultIfFalse>
```

Here, `<test>` is evaluated to obtain a Boolean value, and the result of the operator is either `<resultIfTrue>` or `<resultIfFalse>` based on this value.

You might use this as follows to test the value of an `int` variable called `myInteger`:

```
string resultString = (myInteger < 10) ? "Less than 10"
                                       : "Greater than or equal to 10";
```

The result of the ternary operator is one of two strings, both of which may be assigned to `resultString`. The choice of which string to assign is made by comparing the value of `myInteger` to 10, where a value of less than 10 results in the first string being assigned, and a value of greater than or equal to 10 results in the second string being assigned. For example, if `myInteger` is `4`, then `resultString` will be assigned the string `Less than 10`.

This operator is fine for simple assignments such as this, but it isn't really suitable for executing larger amounts of code based on a comparison. A much better way to do this is to use the `if` statement.

The if Statement

The `if` statement is a far more versatile and useful way to make decisions. Unlike `?:` statements, `if` statements don't have a result (so you can't use them in assignments); instead, you use the statement to conditionally execute other statements.

The simplest use of an `if` statement is as follows, where `<test>` is evaluated (it must evaluate to a Boolean value for the code to compile) and the line of code that follows the statement is executed if `<test>` evaluates to `true`:

```
if (<test>)
   <code executed if <test> is true>;
```

After this code is executed, or if it isn't executed due to `<test>` evaluating to `false`, program execution resumes at the next line of code.

You can also specify additional code using the `else` statement in combination with an `if` statement. This statement is executed if `<test>` evaluates to `false`:

```
if (<test>)
 <code executed if <test> is true>;
else
 <code executed if <test> is false>;
```

Both sections of code can span multiple lines using blocks in braces:

```
if (<test>)
{
   <code executed if <test> is true>;
}
else
{
   <code executed if <test> is false>;
}
```

As a quick example, you could rewrite the code from the last section that used the ternary operator:

```
string resultString = (myInteger < 10) ? "Less than 10"
                                       : "Greater than or equal to 10";
```

Because the result of the `if` statement cannot be assigned to a variable, you have to assign a value to the variable in a separate step:

```
string resultString;
if (myInteger < 10)
   resultString = "Less than 10";
else
   resultString = "Greater than or equal to 10";
```

Code such as this, although more verbose, is far easier to read and understand than the equivalent ternary form, and enables far more flexibility.

The following Try It Out illustrates the use of the `if` statement.

TRY IT OUT Using the if Statement

1. Create a new console application called Ch04Ex02 and save it in the directory `C:\BegVCSharp\Chapter04`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   string comparison;
   Console.WriteLine("Enter a number:");
   double var1 = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("Enter another number:");
   double var2 = Convert.ToDouble(Console.ReadLine());
   if (var1 < var2)
      comparison = "less than";
   else
   {
      if (var1 == var2)
         comparison = "equal to";
      else
          comparison = "greater than";
   }
   Console.WriteLine("The first number is {0} the second number.",
                     comparison);
   Console.ReadKey();
}
```

Code snippet Ch04Ex02\Program.cs

3. Execute the code and enter two numbers at the prompts (see Figure 4-2).

FIGURE 4-2

How It Works

The first section of code is very familiar. It simply obtains two `double` values from user input:

```
string comparison;
Console.WriteLine("Enter a number:");
double var1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());
```

Next, you assign a string to the `string` variable `comparison` based on the values obtained for `var1` and `var2`. First you check whether `var1` is less than `var2`:

```
if (var1 < var2)
   comparison = "less than";
```

If this isn't the case, then `var1` is either greater than or equal to `var2`. In the `else` section of the first comparison, you need to nest a second comparison:

```
else
{
   if (var1 == var2)
      comparison = "equal to";
```

The `else` section of this second comparison is reached only if `var1` is greater than `var2`:

```
   else
       comparison = "greater than";
}
```

Finally, you write the value of `comparison` to the console:

```
Console.WriteLine("The first number is {0} the second number.",
                  comparison);
```

The nesting used here is just one method of performing these comparisons. You could equally have written this:

```
if (var1 < var2)
   comparison = "less than";
if (var1 == var2)
   comparison = "equal to";
if (var1 > var2)
    comparison = "greater than";
```

The disadvantage with this method is that you are performing three comparisons regardless of the values of `var1` and `var2`. With the first method, you perform only one comparison if `var1 < var2` is `true`, and two comparisons otherwise (you also perform the `var1 == var2` comparison), resulting in fewer lines of code being executed. The difference in performance here is slight, but it would be significant in applications where speed of execution is crucial.

Checking More Conditions Using if Statements

In the preceding example, you checked for three conditions involving the value of `var1`. This covered all possible values for this variable. Sometimes, you might want to check for specific values — for example, if `var1` is equal to 1, 2, 3, or 4, and so on. Using code such as the preceding can result in annoyingly nested code:

```
if (var1 == 1)
{
   // Do something.
}
else
{
   if (var1 == 2)
   {
      // Do something else.
   }
   else
   {
      if (var1 == 3 || var1 == 4)
      {
         // Do something else.
      }
      else
      {
         // Do something else.
      }
   }
}
```

COMMON MISTAKES *It's a common mistake to write conditions such as* `if (var1 == 3 || var1 == 4)` *as* `if (var1 == 3 || 4)`*. Here, owing to operator precedence, the* `==` *operator is processed first, leaving the* `||` *operator to operate on a Boolean and a numeric operand, which causes an error.*

In these situations, consider using a slightly different indentation scheme and contracting the section of code for the `else` blocks (that is, using a single line of code after the `else` blocks, rather than a block of code). That way, you end up with a structure involving `else if` statements:

```
if (var1 == 1)
{
   // Do something.
}
```

```
else if (var1 == 2)
{
   // Do something else.
}
else if (var1 == 3 || var1 == 4)
{
   // Do something else.
}
else
{
   // Do something else.
}
```

These `else if` statements are really two separate statements, and the code is functionally identical to the previous code, but much easier to read. When making multiple comparisons such as this, consider using the `switch` statement as an alternative branching structure.

The switch Statement

The `switch` statement is similar to the `if` statement in that it executes code conditionally based on the value of a test. However, `switch` enables you to test for multiple values of a test variable in one go, rather than just a single condition. This test is limited to discrete values, rather than clauses such as "greater than X," so its use is slightly different; but it can be a powerful technique.

The basic structure of a `switch` statement is as follows:

```
switch (<testVar>)
{
   case <comparisonVal1>:
      <code to execute if <testVar> == <comparisonVal1> >
      break;
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal2> >
      break;
   ...
   case <comparisonValN>:
      <code to execute if <testVar> == <comparisonValN> >
      break;
   default:
      <code to execute if <testVar> != comparisonVals>
      break;
}
```

The value in `<testVar>` is compared to each of the `<comparisonValX>` values (specified with `case` statements). If there is a match, then the code supplied for this match is executed. If there is no match, then the code in the `default` section is executed if this block exists.

On completion of the code in each section, you have an additional command, `break`. It is illegal for the flow of execution to reach a second `case` statement after processing one `case` block.

NOTE *The behavior where the flow of execution is forbidden from flowing from one* `case` *block to the next is one area in which C# differs from C++. In C++ the processing of* `case` *statements is allowed to run from one to another.*

The `break` statement here simply terminates the `switch` statement, and processing continues on the statement following the structure.

There are alternative methods of preventing flow from one `case` statement to the next in C# code. You can use the `return` statement, which results in termination of the current function, rather than just the `switch` structure (see Chapter 6 for more details about this), or a `goto` statement. `goto` statements (as detailed earlier) work here because `case` statements actually define labels in C# code. Here is an example:

```
switch (<testVar>)
{
   case <comparisonVal1>:
      <code to execute if <testVar> == <comparisonVal1> >
      goto case <comparisonVal2>;
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal2> >
      break;
   ...
```

One exception to the rule that the processing of one `case` statement can't run freely into the next: If you place multiple `case` statements together (*stack* them) before a single block of code, then you are in effect checking for multiple conditions at once. If any of these conditions is met, then the code is executed. Here's an example:

```
switch (<testVar>)
{
   case <comparisonVal1>:
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal1> or
                          <testVar> == <comparisonVal2> >
      break;
   ...
```

These conditions also apply to the `default` statement. There is no rule stipulating that this statement must be the last in the list of comparisons, and you can stack it with `case` statements if you want. Adding a breakpoint with `break`, `goto`, or `return` ensures that a valid execution path exists through the structure in all cases.

Each of the `<comparisonValX>` comparisons must be a constant value. One way of doing this is to provide literal values, like this:

```
switch (myInteger)
{
   case 1:
      <code to execute if myInteger == 1>
      break;
   case -1:
      <code to execute if myInteger == -1>
      break;
   default:
      <code to execute if myInteger != comparisons>
      break;
}
```

Another way is to use *constant variables*. Constant variables (also known as just "constants," avoiding the oxymoron) are just like any other variable except for one key factor: The value they contain never

changes. Once you assign a value to a constant variable, then that is the value it has for the duration of code execution. Constant variables can come in handy here, because it is often easier to read code where the actual values being compared are hidden from you at the time of comparison.

You declare constant variables using the `const` keyword in addition to the variable type, and you *must* assign them values at this time, as shown here:

```
const int intTwo = 2;
```

The preceding code is perfectly valid, but if you try

```
const int intTwo;
intTwo = 2;
```

you will get an error and won't be able to compile your code. This also happens if you try to change the value of a constant variable through any other means after initial assignment.

The following Try It Out uses a `switch` statement to write different strings to the console, depending on the value you enter for a test string.

TRY IT OUT Using the switch Statement

1. Create a new console application called Ch04Ex03 and save it to the directory `C:\BegVCSharp\Chapter04`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   const string myName = "karli";
   const string sexyName = "angelina";
   const string sillyName = "ploppy";
   string name;
   Console.WriteLine("What is your name?");
   name = Console.ReadLine();
   switch (name.ToLower())
   {
      case myName:
         Console.WriteLine("You have the same name as me!");
         break;
      case sexyName:
         Console.WriteLine("My, what a sexy name you have!");
         break;
      case sillyName:
         Console.WriteLine("That's a very silly name.");
         break;
   }
   Console.WriteLine("Hello {0}!", name);
   Console.ReadKey();
}
```

Code snippet Ch04Ex03\Program.cs

3. Execute the code and enter a name. The result is shown in Figure 4-3.

FIGURE 4-3

How It Works

The code sets up three constant strings, accepts a string from the user, and then writes out text to the console based on the string entered. Here, the strings are names.

When you compare the name entered (in the variable `name`) to your constant values, you first force it into lowercase with `name.ToLower()`. This is a standard command that works with all string variables, and it comes in handy when you're not sure what the user entered. Using this technique, the strings `Karli`, `kArLi`, `karli`, and so on all match the test string `karli`.

The `switch` statement itself attempts to match the string entered with the constant values you have defined, and, if successful, writes out a personalized message to greet the user. If no match is made, you offer a generic greeting.

`switch` statements place no limit on the amount of `case` sections they contain, so you could extend this code to cover every name you can think of should you want . . . but it might take a while!

LOOPING

Looping refers to the repeated execution of statements. This technique comes in very handy because it means that you can repeat operations as many times as you want (thousands, even millions, of times) without having to write the same code each time.

As a simple example, consider the following code for calculating the amount of money in a bank account after 10 years, assuming that interest is paid each year and no other money flows into or out of the account:

```
double balance = 1000;
double interestRate = 1.05; // 5% interest/year
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

```
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

Writing the same code 10 times seems a bit wasteful, and what if you wanted to change the duration from 10 years to some other value? You'd have to manually copy the line of code the required amount of times, which would be a bit of a pain! Luckily, you don't have to do this. Instead, you can have a loop that executes the instruction you want the required number of times.

Another important type of loop is one in which you loop until a certain condition is fulfilled. These loops are slightly simpler than the situation detailed previously (although no less useful), so they're a good starting point.

do Loops

`do` loops operate as follows. The code you have marked out for looping is executed, a Boolean test is performed, and the code executes again if this test evaluates to `true`, and so on. When the test evaluates to `false`, the loop exits.

The structure of a `do` loop is as follows, where *<Test>* evaluates to a Boolean value:

```
do
{
   <code to be looped>
} while (<Test>);
```

NOTE *The semicolon after the* `while` *statement is required.*

For example, you could use the following to write the numbers from 1 to 10 in a column:

```
int i = 1;
do
{
   Console.WriteLine("{0}", i++);
} while (i <= 10);
```

Here, you use the suffix version of the `++` operator to increment the value of `i` after it is written to the screen, so you need to check for `i <= 10` to include 10 in the numbers written to the console.

The following Try It Out uses this for a slightly modified version of the code shown earlier, where you calculated the balance in an account after 10 years. Here, you use a loop to calculate how many years it will take to get a specified amount of money in the account, based on a starting amount and a fixed interest rate.

TRY IT OUT Using do Loops

1. Create a new console application called Ch04Ex04 and save it to the directory `C:\BegVCSharp\Chapter04`.

2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
 {
    double balance, interestRate, targetBalance;
    Console.WriteLine("What is your current balance?");
    balance = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("What is your current annual interest rate (in %)?");
    interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
    Console.WriteLine("What balance would you like to have?");
    targetBalance = Convert.ToDouble(Console.ReadLine());
    int totalYears = 0;
    do
    {
       balance *= interestRate;
       ++totalYears;
    }
    while (balance < targetBalance);
    Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                      totalYears, totalYears == 1 ? "": "s", balance);
    Console.ReadKey();
 }
```

Code snippet Ch04Ex04\Program.cs

3. Execute the code and enter some values. A sample result is shown in Figure 4-4.

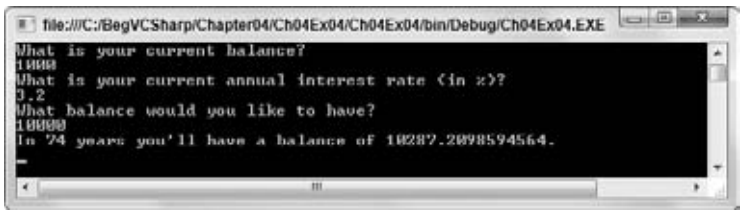

FIGURE 4-4

How It Works

This code simply repeats the simple annual calculation of the balance with a fixed interest rate as many times as is necessary for the balance to satisfy the terminating condition. You keep a count of how many years have been accounted for by incrementing a counter variable with each loop cycle:

```
int totalYears = 0;
 do
 {
    balance *= interestRate;
    ++totalYears;
 }
 while (balance < targetBalance);
```

You can then use this counter variable as part of the result output:

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "": "s", balance);
```

NOTE *Perhaps the most common usage of the* `?:` *(ternary) operator is to conditionally format text with the minimum of code. Here, you output an "s" after "year" if* `totalYears` *isn't equal to 1.*

Unfortunately, this code isn't perfect. Consider what happens when the target balance is less than the current balance. The output will be similar to what is shown in Figure 4-5.

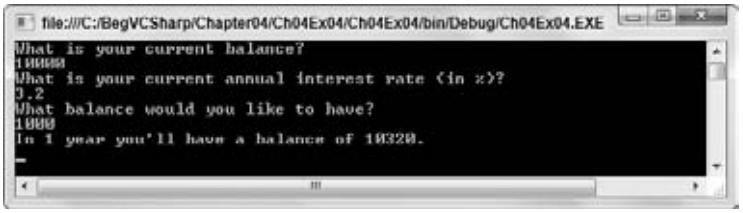

FIGURE 4-5

`do` loops always execute at least once. Sometimes, as in this situation, this isn't ideal. Of course, you could add an `if` statement:

```
int totalYears = 0;
if (balance < targetBalance)
{
   do
   {
      balance *= interestRate;
      ++totalYears;
   }
   while (balance < targetBalance);
}
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "": "s", balance);
```

Clearly, this adds unnecessary complexity. A far better solution is to use a `while` loop.

while Loops

`while` loops are very similar to `do` loops, but they have one important difference: The Boolean test in a `while` loop takes place at the start of the loop cycle, not at the end. If the test evaluates to `false`, then the loop cycle is never executed. Instead, program execution jumps straight to the code following the loop.

Here's how `while` loops are specified:

```
while (<Test>)
{
   <code to be looped>
}
```

They can be used in almost the same way as `do` loops:

```
int i = 1;
while (i <= 10)
{
   Console.WriteLine("{0}", i++);
}
```

This code has the same result as the `do` loop shown earlier; it outputs the numbers 1 to 10 in a column. The following Try It Out demonstrates how you can modify the last example to use a `while` loop.

TRY IT OUT Using while Loops

1. Create a new console application called Ch04Ex05 and save it to the directory `C:\BegVCSharp\Chapter04`.
2. Modify the code as follows (use the code from Ch04Ex04 as a starting point, and remember to delete the `while` statement at the end of the original `do` loop):


```
static void Main(string[] args)
{
   double balance, interestRate, targetBalance;
   Console.WriteLine("What is your current balance?");
   balance = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("What is your current annual interest rate (in %)?");
   interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
   Console.WriteLine("What balance would you like to have?");
   targetBalance = Convert.ToDouble(Console.ReadLine());
   int totalYears = 0;
   while (balance < targetBalance)
   {
      balance *= interestRate;
      ++totalYears;
   }
   Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                     totalYears, totalYears == 1 ? "": "s", balance);
      if (totalYears == 0)
         Console.WriteLine(
            "To be honest, you really didn't need to use this calculator.");
   Console.ReadKey();
}
```

Code snippet Ch04Ex05\Program.cs

3. Execute the code again, but this time use a target balance that is less than the starting balance, as shown in Figure 4-6.

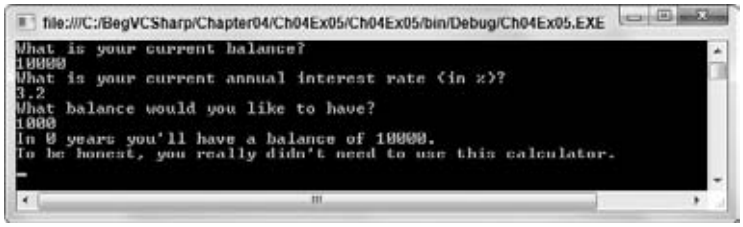

FIGURE 4-6

How It Works

This simple change from a `do` loop to a `while` loop has solved the problem in the last example. By moving the Boolean test to the beginning, you provide for the circumstance where no looping is required, and you can jump straight to the result.

Of course, other alternatives are possible in this situation. For example, you could check the user input to ensure that the target balance is greater than the starting balance. In that case, you can place the user input section in a loop as follows:

```
Console.WriteLine("What balance would you like to have?");
do
{
   targetBalance = Convert.ToDouble(Console.ReadLine());
   if (targetBalance <= balance)
      Console.WriteLine("You must enter an amount greater than " +
                "your current balance!\nPlease enter another value.");
}
while (targetBalance <= balance);
```

This rejects values that don't make sense, so the output looks like Figure 4-7.

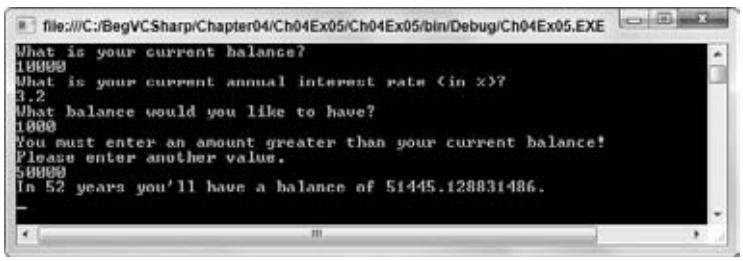

FIGURE 4-7

This *validation* of user input is an important topic when it comes to application design, and many examples of it appear throughout this book.

for Loops

The last type of loop to look at in this chapter is the `for` loop. This type of loop executes a set number of times and maintains its own counter. To define a `for` loop you need the following information:

- A starting value to initialize the counter variable
- A condition for continuing the loop, involving the counter variable
- An operation to perform on the counter variable at the end of each loop cycle

For example, if you want a loop with a counter that increments from 1 to 10 in steps of one, then the starting value is 1; the condition is that the counter is less than or equal to 10; and the operation to perform at the end of each cycle is to add 1 to the counter.

This information must be placed into the structure of a `for` loop as follows:

```
for (<initialization>; <condition>; <operation>)
{
   <code to loop>
}
```

This works exactly the same way as the following `while` loop:

```
<initialization>
while (<condition>)
{
   <code to loop>
   <operation>
}
```

The format of the `for` loop makes the code easier to read, however, because the syntax involves the complete specification of the loop in one place, rather than dividing it over several statements in different areas of the code.

Earlier, you used `do` and `while` loops to write out the numbers from 1 to 10. The code that follows shows what is required to do this using a `for` loop:

```
int i;
for (i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

The counter variable, an integer called `i`, starts with a value of `1` and is incremented by `1` at the end of each cycle. During each cycle, the value of `i` is written to the console.

When the code resumes after the loop, `i` has a value of `11`. That's because at the end of the cycle where `i` is equal to 10, `i` is incremented to `11`. This happens before the condition `i <= 10` is processed, at which point the loop ends. As with `while` loops, `for` loops execute only if the condition evaluates to `true` before the first cycle, so the code in the loop doesn't necessarily run at all.

As a final note, you can declare the counter variable as part of the `for` statement, rewriting the preceding code as follows:

```
for (int i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

If you do this, though, the variable `i` won't be accessible from code outside this loop (see the section "Variable Scope" in Chapter 6).

The next Try It Out uses `for` loops, and because you have already used several loops now, this example is a bit more interesting: It displays a Mandelbrot set (but using plain-text characters, so it won't look that spectacular).

TRY IT OUT Using for Loops

1. Create a new console application called Ch04Ex06 and save it to the directory `C:\BegVCSharp\Chapter04`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   double realCoord, imagCoord;
   double realTemp, imagTemp, realTemp2, arg;
   int iterations;
   for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
   {
      for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
      {
         iterations = 0;
         realTemp = realCoord;
         imagTemp = imagCoord;
         arg = (realCoord * realCoord) + (imagCoord * imagCoord);
         while ((arg < 4) && (iterations < 40))
         {
            realTemp2 = (realTemp * realTemp)-(imagTemp * imagTemp)
                  -realCoord;
            imagTemp = (2 * realTemp * imagTemp) -imagCoord;
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
         }
         switch (iterations % 4)
         {
            case 0:
               Console.Write(".");
               break;
            case 1:
               Console.Write("o");
               break;
            case 2:
               Console.Write("O");
               break;
```

```
                case 3:
                   Console.Write("@");
                   break;
             }
          }
          Console.Write("\n");
       }
       Console.ReadKey();
    }
```

Code snippet Ch04Ex06\Program.cs

3. Execute the code. The result is shown in Figure 4-8.

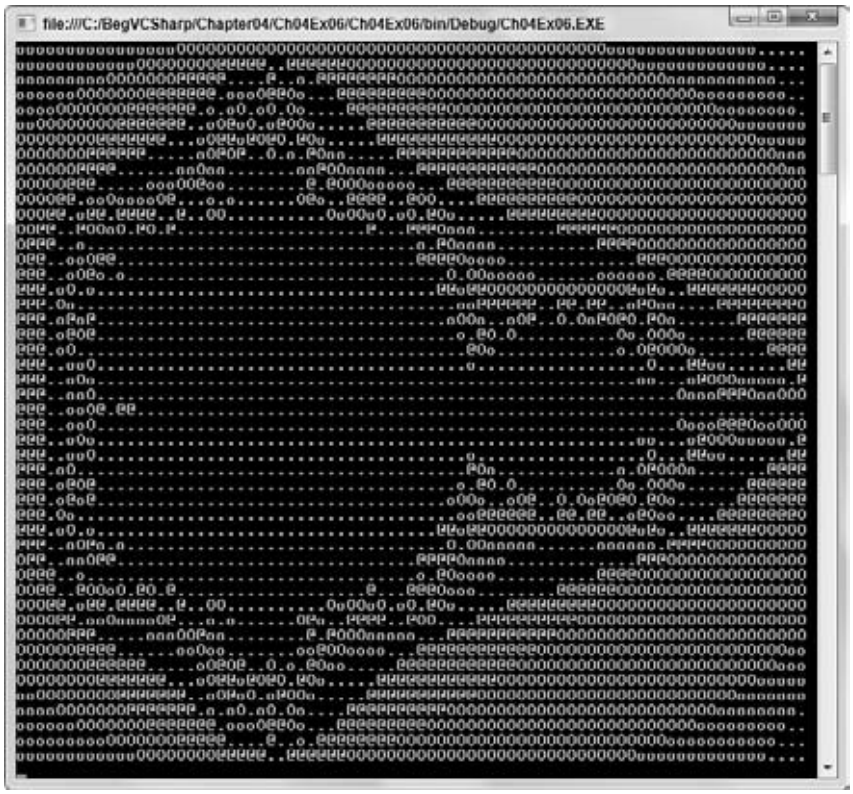

FIGURE 4-8

How It Works

Details about calculating Mandelbrot sets are beyond the scope of this chapter, but you should understand why you need the loops used in this code. Feel free to skim through the following two paragraphs if the mathematics doesn't interest you; it's an understanding of the code that is important here.

Each position in a Mandelbrot image corresponds to an imaginary number of the form `N = x + y*i`, where the real part is `x`, the imaginary part is `y`, and `i` is the square root of `-1`. The `x` and `y` coordinates of the position in the image correspond to the `x` and `y` parts of the imaginary number.

For each position on the image, you look at the argument of `N`, which is the square root of `x*x + y*y`. If this value is greater than or equal to 2, you say that the position corresponding to this number has a value of 0. If the argument of `N` is less than 2, you change `N` to a value of `N*N-N` (giving you `N = (x*x-y*y-x) + (2*x*y-y)*i)` and check the argument of this new value of `N` again. If this value is greater than or equal to 2, you say that the position corresponding to this number has a value of 1. This process continues until you either assign a value to the position on the image or perform more than a certain number of iterations.

Based on the values assigned to each point in the image, you would, in a graphical environment, place a pixel of a certain color on the screen. However, because you are using a text display, you simply place characters onscreen instead.

Now, back to the code, and the loops contained in it. You begin by declaring the variables you need for your calculation:

```
double realCoord, imagCoord;
double realTemp, imagTemp, realTemp2, arg;
int iterations;
```

Here, `realCoord` and `imagCoord` are the real and imaginary parts of `N`, and the other `double` variables are for temporary information during computation. `iterations` records how many iterations it takes before the argument of `N` (`arg`) is 2 or greater.

Next, you start two `for` loops to cycle through coordinates covering the whole of the image (using a slightly more complex syntax for modifying your counters than `++` or `--`, a common and powerful technique):

```
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
{
   for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
   {
```

Here, appropriate limits have been used to show the main section of the Mandelbrot set. Feel free to play around with these if you want to try "zooming in" on the image.

Within these two loops you have code that pertains to a single point in the Mandelbrot set, giving you a value for `N` to play with. This is where you perform your calculation of iterations required, giving you a value to plot for the current point.

First, initialize some variables:

```
iterations = 0;
realTemp = realCoord;
imagTemp = imagCoord;
arg = (realCoord * realCoord) + (imagCoord * imagCoord);
```

Next, you have a `while` loop to perform your iterating. Use a `while` loop rather than a `do` loop, in case the initial value of `N` has an argument greater than 2 already, in which case `iterations == 0` is the answer you are looking for and no further calculations are necessary.

Note that you're not quite calculating the argument fully here. You're just getting the value of `x*x + y*y` and checking whether that value is less than 4. This simplifies the calculation, because you know that 2 is the square root of 4 and don't have to calculate any square roots yourself:

```
while ((arg < 4) && (iterations < 40))
{
   realTemp2 = (realTemp * realTemp)-(imagTemp * imagTemp)
         -realCoord;
   imagTemp = (2 * realTemp * imagTemp)-imagCoord;
```

```
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
         }
```

The maximum number of iterations of this loop, which calculates values as detailed above, is 40.

Once you have a value for the current point stored in `iterations`, you use a `switch` statement to choose a character to output. You just use four different characters here, instead of the 40 possible values, and use the modulus operator (`%`) so that values of 0, 4, 8, and so on provide one character; values of 1, 5, 9, and so on provide another character, and so forth:

```
switch (iterations % 4)
{
   case 0:
      Console.Write(".");
      break;
   case 1:
      Console.Write("o");
      break;
   case 2:
      Console.Write("O");
      break;
   case 3:
      Console.Write("@");
      break;
}
```

You use `Console.Write()` here, rather than `Console.WriteLine()`, because you don't want to start a new line every time you output a character. At the end of one of the innermost `for` loops, you do want to end a line, so you simply output an end-of-line character using the escape sequence shown earlier:

```
      }
      Console.Write("\n");
   }
```

This results in each row being separated from the next and lining up appropriately. The final result of this application, though not spectacular, is fairly impressive, and certainly shows how useful looping and branching can be.

Interrupting Loops

Sometimes you want finer-grained control over the processing of looping code. C# provides four commands to help you here, three of which were shown in other situations:

- `break` — Causes the loop to end immediately
- `continue` — Causes the current loop cycle to end immediately (execution continues with the next loop cycle)
- `goto` — Allows jumping out of a loop to a labeled position (not recommended if you want your code to be easy to read and understand)
- `return` — Jumps out of the loop and its containing function (see Chapter 6)

The `break` command simply exits the loop, and execution continues at the first line of code after the loop, as shown in the following example:

```
int i = 1;
while (i <= 10)
{
   if (i == 6)
      break;
   Console.WriteLine("{0}", i++);
}
```

This code writes out the numbers from 1 to 5 because the `break` command causes the loop to exit when `i` reaches 6.

`continue` only stops the current cycle, not the whole loop, as shown here:

```
int i;
for (i = 1; i <= 10; i++)
{
   if ((i % 2) == 0)
      continue;
   Console.WriteLine(i);
}
```

In the preceding example, whenever the remainder of `i` divided by 2 is zero, the `continue` statement stops the execution of the current cycle, so only the numbers `1`, `3`, `5`, `7`, and `9` are displayed.

The third method of interrupting a loop is to use `goto`, as shown earlier:

```
int i = 1;
while (i <= 10)
{
   if (i == 6)
      goto exitPoint;
   Console.WriteLine("{0}", i++);
}
Console.WriteLine("This code will never be reached.");
exitPoint:
Console.WriteLine("This code is run when the loop is exited using goto.");
```

Note that exiting a loop with `goto` is legal (if slightly messy), but it is illegal to use `goto` to jump into a loop from outside.

Infinite Loops

It is possible, through both coding errors and design, to define loops that never end, so-called *infinite loops*. As a very simple example, consider the following:

```
while (true)
{
   // code in loop
}
```

This can be useful, and you can always exit such loops using code such as `break` statements or manually by using the Windows Task Manager. However, when this occurs by accident, it can be annoying. Consider the following loop, which is similar to the `for` loop in the previous section:

```
int i = 1;
while (i <= 10)
{
   if ((i % 2) == 0)
      continue;
   Console.WriteLine("{0}", i++);
}
```

Here, `i` isn't incremented until the last line of code in the loop, which occurs after the `continue` statement. If this `continue` statement is reached (which it will be when `i` is 2), the next loop cycle will be using the same value of `i`, continuing the loop, testing the same value of `i`, continuing the loop, and so on. This will cause the application to freeze. Note that it's still possible to quit the frozen application in the normal way, so you won't have to reboot if this happens.

SUMMARY

In this chapter, you increased your programming knowledge by considering various structures that you can use in your code. The proper use of these structures is essential when you start making more complex applications, and you will see them used throughout this book.

You first spent some time looking at Boolean logic, with a bit of bitwise logic thrown in for good measure. Looking back on this after working through the rest of the chapter should confirm the suggestion that this topic is very important when it comes to implementing branching and looping code in your programs. It is essential to become very familiar with the operators and techniques detailed in this section.

Branching enables you to conditionally execute code, which, when combined with looping, enables you to create convoluted structures in your C# code. When you have loops inside loops inside `if` structures inside loops, you start to see why code indentation is so useful! If you shift all your code to the left of the screen, it instantly becomes difficult to parse by eye, and even more difficult to debug. Make sure you've got the hang of indentation at this stage — you'll appreciate it later! VS does a lot of this for you, but it's a good idea to indent code as you type it anyway.

The next chapter covers variables in more depth.

EXERCISES

1. If you have two integers stored in variables `var1` and `var2`, what Boolean test can you perform to determine whether one or the other (but not both) is greater than 10?

2. Write an application that includes the logic from Exercise 1, obtains two numbers from the user, and displays them, but rejects any input where both numbers are greater than 10 and asks for two new numbers.

continues

3. What is wrong with the following code?

```
int i;
for (i = 1; i <= 10; i++)
{
   if ((i % 2) = 0)
      continue;
   Console.WriteLine(i);
}
```

4. Modify the Mandelbrot set application to request image limits from the user and display the chosen section of the image. The current code outputs as many characters as will fit on a single line of a console application; consider making every image chosen fit in the same amount of space to maximize the viewable area.

Answers to Exercises can be found in Appendix A.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Boolean logic	Boolean logic involves using Boolean (true or false) values to evaluate conditions. Boolean operators are used to perform comparisons between values and return Boolean results. Some Boolean operators are also used to perform bitwise operations on the underlying bit structure of values, and there are some specialized bitwise operators too.
Branching	You can use Boolean logic to control program flow. The result of an expression that evaluates to a Boolean value can be used to determine whether a block of code is executed. You do this with `if` statements or the `?:` (ternary) operator for simple branching, or the `switch` statement to check multiple conditions simultaneously.
Looping	Looping allows you to execute blocks of code a number of times according to conditions you specify. You can use `do` and `while` loops to execute code while a Boolean expression evaluates to `true`, and `for` loops to include a counter in your looping code. Loops can be interrupted by cycle (with `continue`) or completely (with `break`). Some loops only end if you interrupt them; these are called infinite loops.

5

More About Variables

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to perform implicit and explicit conversions between types
- How to create and use enum types
- How to create and use struct types
- How to create and use arrays
- How to manipulate string values

Now that you've seen a bit more of the C# language, let's go back and tackle some of the more involved topics concerning variables.

The first subject you look at in this chapter is *type conversion*, whereby you convert values from one type into another. You've already seen a bit of this, but you look at it formally here. A grasp of this topic gives you a greater understanding of what happens when you mix types in expressions (intentionally or unintentionally) as well as tighter control over the way that data is manipulated. This helps you to streamline your code and avoid nasty surprises.

Then you'll look at a few more types of variables that you can use:

- **Enumerations**: Variable types that have a user-defined discrete set of possible values that can be used in a human-readable way.
- **Structs:** Composite variable types made up of a user-defined set of other variable types.
- **Arrays**: Types that hold multiple variables of one type, allowing index access to the individual value.

These are slightly more complex than the simple types you've been using up to now, but they can make your life much easier. Finally, you'll explore another useful subject concerning strings: basic string manipulation.

TYPE CONVERSION

Earlier in this book you saw that all data, regardless of type, is simply a sequence of bits — that is, a sequence of zeros and ones. The meaning of the variable is determined by the way in which this data is interpreted. The simplest example of this is the `char` type. This type represents a character in the Unicode character set using a number. In fact, the number is stored in exactly the same way as a `ushort` — both of them store a number between 0 and 65535.

However, in general, the different types of variables use varying schemes to represent data. This implies that even if it were possible to place the sequence of bits from one variable into a variable of a different type (perhaps they use the same amount of storage, or perhaps the target type has enough storage space to include all the source bits), the results might not be what you expect.

Instead of this one-to-one mapping of bits from one variable into another, you need to use *type conversion* on the data. Type conversion takes two forms:

- **Implicit conversion:** Conversion from type A to type B is possible in all circumstances, and the rules for performing the conversion are simple enough for you to trust in the compiler.
- **Explicit conversion:** Conversion from type A to type B is possible only in certain circumstances or where the rules for conversion are complicated enough to merit additional processing of some kind.

Implicit Conversions

Implicit conversion requires no work on your part and no additional code. Consider the code shown here:

```
var1 = var2;
```

This assignment may involve an implicit conversion if the type of `var2` can be implicitly converted into the type of `var1`, but it could just as easily involve two variables with the same type, in which case no implicit conversion is necessary. For example, the values of `ushort` and `char` are effectively interchangeable, because both store a number between 0 and 65535. You can convert values between these types implicitly, as illustrated by the following code:

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

Here, the value stored in `sourceVar` is placed in `destinationVar`. When you output the variables with the two `Console.WriteLine()` commands, you get the following output:

```
sourceVar val: a
destinationVar val: 97
```

Even though the two variables store the same information, they are interpreted in different ways using their type.

There are many implicit conversions of simple types; `bool` and `string` have no implicit conversions, but the numeric types have a few. For reference, the following table shows the numeric conversions that the compiler can perform implicitly (remember that `char`s are stored as numbers, so `char` counts as a numeric type).

TYPE	CAN SAFELY BE CONVERTED TO
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Don't worry — you don't need to learn this table by heart, because it's actually quite easy to work out which conversions the compiler can do implicitly. Back in Chapter 3, you saw a table showing the range of possible values for every simple numeric type. The implicit conversion rule for these types is this: Any type *A* whose range of possible values completely fits inside the range of possible values of type *B* can be implicitly converted into that type.

The reasoning for this is simple. If you try to fit a value into a variable but that value is outside the range of values the variable can take, then there will be a problem. For example, a `short` type variable is capable of storing values up to 32767, and the maximum value allowed into a `byte` is 255, so there could be problems if you try to convert a `short` value into a `byte` value. If the `short` holds a value between 256 and 32767, then it simply won't fit into a `byte`.

If you know that the value in your `short` type variable is less than 255, then you should be able to convert the value, right? The simple answer is that, of course, you can. The slightly more complex answer is that, of course, you can, but you must use an *explicit* conversion. Performing an explicit conversion is a bit like saying "OK, I know you've warned me about doing this, but I'll take responsibility for what happens."

Explicit Conversions

As the name suggests, an explicit conversion occurs when you explicitly ask the compiler to convert a value from one data type to another. These conversions require extra code, and the format of this code may vary, depending on the exact conversion method. Before you look at any of this explicit conversion code, look at what happens if you *don't* add any.

For example, the following modification to the code from the last section attempts to convert a `short` value into a `byte`:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

If you attempt to compile the preceding code, you will receive the following error:

```
Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists
(are you missing a cast?)
```

Luckily for you, the C# compiler can detect missing explicit conversions!

To get this code to compile, you need to add the code to perform an explicit conversion. The easiest way to do that in this context is to *cast* the `short` variable into a `byte` (as suggested by the preceding error string). Casting basically means forcing data from one type into another, and it uses the following simple syntax:

```
<(destinationType)sourceVar>
```

This will convert the value in `<sourceVar>` into `<destinationType>`.

NOTE *Casting is only possible in some situations. Types that bear little or no relation to each other are likely not to have casting conversions defined.*

You can, therefore, modify your example using this syntax to force the conversion from a `short` to a `byte`:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

This results in the following output:

```
sourceVar val: 7
destinationVar val: 7
```

What happens when you try to force a value into a variable into which it won't fit? Modifying your code as follows illustrates this:

```
byte destinationVar;
short sourceVar = 281;
```

```
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

This results in the following:

```
sourceVar val: 281
destinationVar val: 25
```

What happened? Well, look at the binary representations of these two numbers, along with the maximum value that can be stored in a byte, which is 255:

```
281 = 100011001
 25 = 000011001
255 = 011111111
```

You can see that the leftmost bit of the source data has been lost. This immediately raises a question: How can you tell when this happens? Obviously, there will be times when you will need to explicitly cast one type into another, and it would be nice to know if any data has been lost along the way. Not detecting this could cause serious errors — for example, in an accounting application or an application determining the trajectory of a rocket to the moon.

One way to do this is simply to check the value of the source variable and compare it with the known limits of the destination variable. Another technique is to force the system to pay special attention to the conversion at runtime. Attempting to fit a value into a variable when that value is too big for the type of that variable results in an *overflow,* and this is the situation you want to check for.

Two keywords exist for setting what is called the *overflow checking context* for an expression: `checked` and `unchecked`. You use these in the following way:

```
checked(<expression>)
unchecked(<expression>)
```

You can force overflow checking in the last example:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

When this code is executed, it will crash with the error message shown in Figure 5-1 (this was compiled in a project called OverflowCheck).

However, if you replace `checked` with `unchecked` in this code, you get the result shown earlier, and no error occurs. That is identical to the default behavior, also shown earlier.

You also can configure your application to behave as if every expression of this type includes the `checked` keyword, unless that expression explicitly uses the `unchecked` keyword (in other words, you can change the default setting for overflow checking). To do this, you modify the properties for your project by right-clicking on it in the Solution Explorer window and selecting the Properties option. Click Build on the left side of the window to bring up the Build settings, as shown in Figure 5-2.

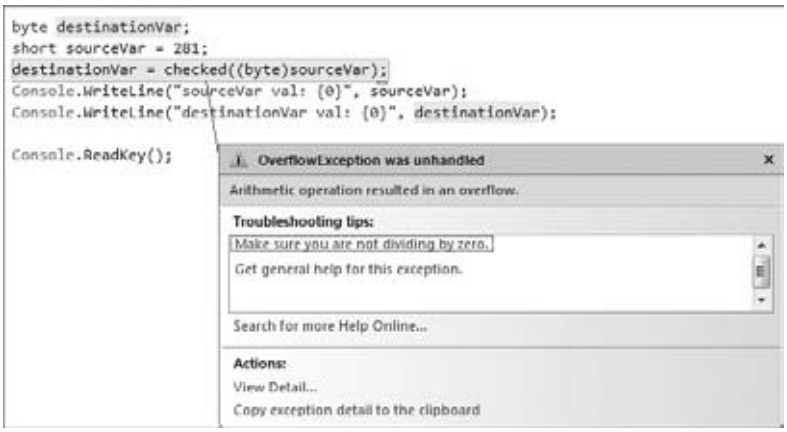

FIGURE 5-1

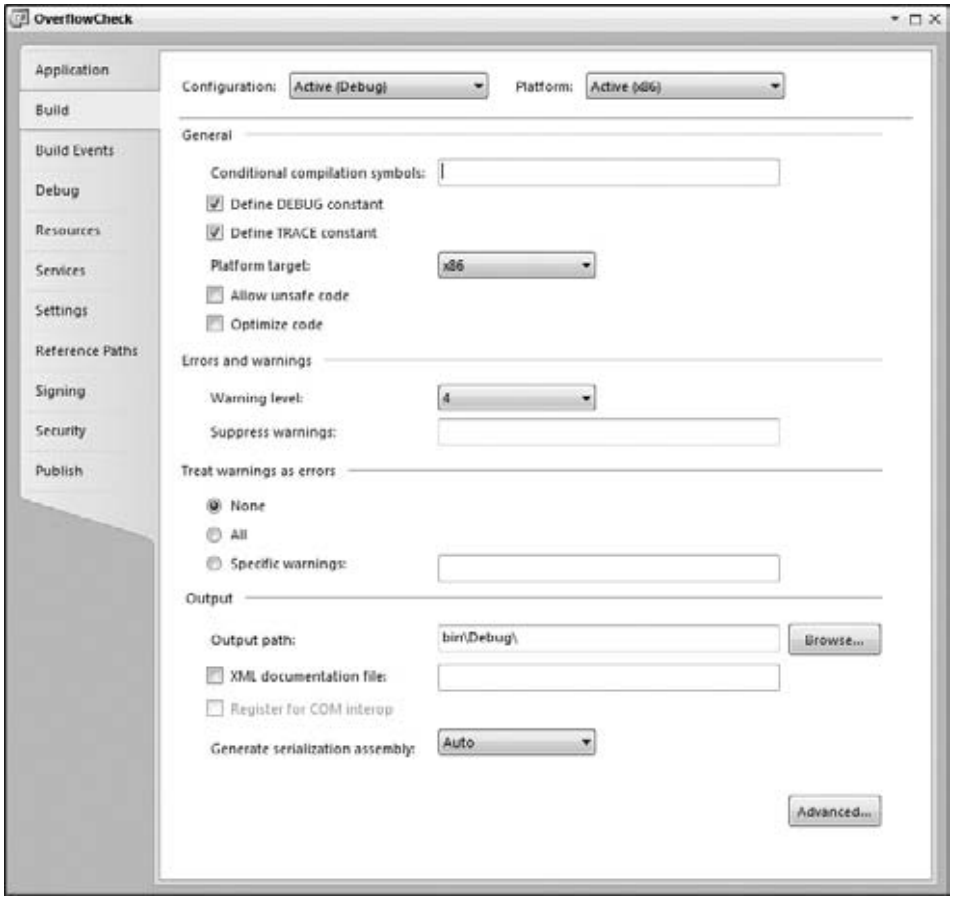

FIGURE 5-2

The property you want to change is one of the Advanced settings, so click the Advanced button. In the dialog that appears, enable the Check for Arithmetic Overflow/Underflow option, as shown in Figure 5-3. By default, this setting is disabled; enabling it provides the `checked` behavior detailed previously.

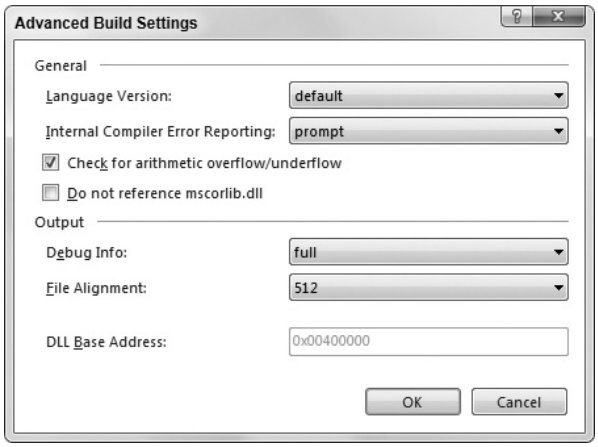

FIGURE 5-3

Explicit Conversions Using the Convert Commands

The type of explicit conversion you have been using in many of the Try It Out examples in this book is a bit different from those you have seen so far in this chapter. You have been converting string values into numbers using commands such as `Convert.ToDouble()`, which is obviously something that won't work for every possible string.

If, for example, you try to convert a string like `Number` into a double value using `Convert.ToDouble()`, you will see the dialog shown in Figure 5-4 when you execute the code.

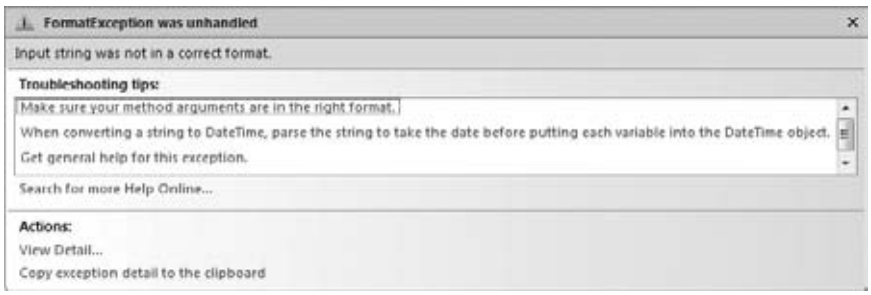

FIGURE 5-4

As you can see, the operation fails. For this type of conversion to work, the string supplied *must* be a valid representation of a number, and that number must be one that won't cause an overflow. A valid representation of a number is one that contains an optional sign (that is, plus or minus), zero or more

digits, an optional period followed by one or more digits, and an optional "e" or "E" followed by an optional sign, one or more digits, and nothing else except spaces (before or after this sequence). Using all of these optional extras, you can recognize strings as complex as `-1.2451e-24` as being a number.

You can specify many such explicit conversions in this way, as the following table shows:

COMMAND	RESULT
`Convert.ToBoolean(val)`	`val` converted to `bool`
`Convert.ToByte(val)`	`val` converted to `byte`
`Convert.ToChar(val)`	`val` converted to `char`
`Convert.ToDecimal(val)`	`val` converted to `decimal`
`Convert.ToDouble(val)`	`val` converted to `double`
`Convert.ToInt16(val)`	`val` converted to `short`
`Convert.ToInt32(val)`	`val` converted to `int`
`Convert.ToInt64(val)`	`val` converted to `long`
`Convert.ToSByte(val)`	`val` converted to `sbyte`
`Convert.ToSingle(val)`	`val` converted to `float`
`Convert.ToString(val)`	`val` converted to `string`
`Convert.ToUInt16(val)`	`val` converted to `ushort`
`Convert.ToUInt32(val)`	`val` converted to `uint`
`Convert.ToUInt64(val)`	`val` converted to `ulong`

Here, `val` can be most types of variable (if it's a type that can't be handled by these commands, the compiler will tell you).

Unfortunately, as the table shows, the names of these conversions are slightly different from the C# type names; for example, to convert to an `int` you use `Convert.ToInt32()`. That's because these commands come from the .NET Framework `System` namespace, rather than being native C#. This enables them to be used from other .NET-compatible languages besides C#.

The important thing to note about these conversions is that they are *always* overflow-checked, and the `checked` and `unchecked` keywords and project property settings have no effect.

The next Try It Out is an example that covers many of the conversion types from this section. It declares and initializes a number of variables of different types and then converts between them implicitly and explicitly.

TRY IT OUT Type Conversions in Practice

1. Create a new console application called Ch05Ex01 and save it in the directory `C:\BegVCSharp\Chapter05`.

2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
    short  shortResult, shortVal = 4;
    int    integerVal = 67;
    long   longResult;
    float  floatVal = 10.5F;
    double doubleResult, doubleVal = 99.999;
    string stringResult, stringVal = "17";
    bool   boolVal = true;

    Console.WriteLine("Variable Conversion Examples\n");

    doubleResult = floatVal * shortVal;
    Console.WriteLine("Implicit, -> double: {0} * {1} -> {2}", floatVal,
       shortVal, doubleResult);

    shortResult = (short)floatVal;
    Console.WriteLine("Explicit, -> short:  {0} -> {1}", floatVal,
       shortResult);

    stringResult = Convert.ToString(boolVal) +
       Convert.ToString(doubleVal);
    Console.WriteLine("Explicit, -> string: \"{0}\" + \"{1}\" -> {2}",
       boolVal, doubleVal, stringResult);

    longResult = integerVal + Convert.ToInt64(stringVal);
    Console.WriteLine("Mixed,    -> long:   {0} + {1} -> {2}",
       integerVal, stringVal, longResult);
    Console.ReadKey();
}
```

Code snippet Ch05Ex01\Program.cs

3. Execute the code. The result is shown in Figure 5-5.

FIGURE 5-5

How It Works

This example contains all of the conversion types you've seen so far — both in simple assignments, as in the short code examples in the preceding discussion, and in expressions. You need to consider both cases because the processing of *every* non-unary operator may result in type conversions, not just assignment operators. For example, the following multiplies a `short` value by a `float` value:

```
shortVal * floatVal
```

In situations such as this, where no explicit conversion is specified, implicit conversion will be used if possible. In this example, the only implicit conversion that makes sense is to convert the `short` into a `float` (as converting a `float` into a `short` requires explicit conversion), so this is the one that will be used.

However, you can override this behavior should you want, as shown here:

```
shortVal * (short)floatVal
```

NOTE *Interestingly, multiplying two* `short` *values together doesn't return a* `short` *value. Because the result of this operation is quite likely to exceed 32767 (the maximum value a* `short` *can hold), it actually returns an* `int`*.*

Explicit conversions performed using this casting syntax take the same operator precedence as other unary operators (such as `++` used as a prefix) — that is, the highest level of precedence.

When you have statements involving mixed types, conversions occur as each operator is processed, according to operator precedence. This means that "intermediate" conversions may occur:

```
doubleResult = floatVal + (shortVal * floatVal);
```

The first operator to be processed here is `*`, which, as discussed previously, will result in `shortVal` being converted to a `float`. Next, you process the `+` operator, which won't require any conversion because it acts on two `float` values (`floatVal` and the `float` type result of `shortVal * floatVal`). Finally, the `float` result of this calculation is converted into a `double` when the `=` operator is processed.

The conversion process can seem complex at first glance, but as long as you break expressions down into parts by taking the operator precedence order into account, you should be able to work things out.

COMPLEX VARIABLE TYPES

In addition to all the simple variable types, C# also offers three slightly more complex (but very useful) sorts of variable: enumerations (often referred to as enums), structs (occasionally referred to as structures), and arrays.

Enumerations

Each of the types you've seen so far (with the exception of `string`) has a clearly defined set of allowed values. Admittedly, this set is so large in types such as `double` that it can practically be considered a

continuum, but it *is* a fixed set nevertheless. The simplest example of this is the `bool` type, which can take only one of two values: `true` or `false`.

There are many other circumstances in which you might want to have a variable that can take one of a fixed set of results. For example, you might want to have an `orientation` type that can store one of the values `north`, `south`, `east`, or `west`.

In situations like this, *enumerations* can be very useful. Enumerations do exactly what you want in this `orientation` type: They allow the definition of a type that can take one of a finite set of values that you supply. What you need to do, then, is create your own enumeration type called `orientation` that can take one of the four possible values.

Note that there is an additional step involved here — you don't just declare a variable of a given type; you declare and detail a user-defined type and then declare a variable of this new type.

Defining Enumerations

You can use the `enum` keyword to define enumerations as follows:

```
enum <typeName>
{
   <value1>,
   <value2>,
   <value3>,
   ...
   <valueN>
}
```

Next, you can declare variables of this new type as follows:

```
<typeName> <varName>;
```

You can assign values using the following:

```
<varName> = <typeName>.<value>;
```

Enumerations have an *underlying type* used for storage. Each of the values that an enumeration type can take is stored as a value of this underlying type, which by default is `int`. You can specify a different underlying type by adding the type to the enumeration declaration:

```
enum <typeName> : <underlyingType>
{
   <value1>,
   <value2>,
   <value3>,
   ...
   <valueN>
}
```

Enumerations can have underlying types of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`.

By default, each value is assigned a corresponding underlying type value automatically according to the order in which it is defined, starting from zero. This means that `<value1>` gets the value 0, `<value2>`

gets 1, <value3> gets 2, and so on. You can override this assignment by using the = operator and specifying actual values for each enumeration value:

```
enum <typeName> : <underlyingType>
{
   <value1> = <actualVal1>,
   <value2> = <actualVal2>,
   <value3> = <actualVal3>,
   ...
   <valueN> = <actualValN>
}
```

In addition, you can specify identical values for multiple enumeration values by using one value as the underlying value of another:

```
enum <typeName> : <underlyingType>
{
   <value1> = <actualVal1>,
   <value2> = <value1>,
   <value3>,
   ...
   <valueN> = <actualValN>
}
```

Any values left unassigned are given an underlying value automatically, whereby the values used are in a sequence starting from 1 greater than the last explicitly declared one. In the preceding code, for example, <value3> will get the value <value1> + 1.

Note that this can cause problems, with values specified after a definition such as <value2> = <value1> being identical to other values. For example, in the following code <value4> will have the same value as <value2>:

```
enum <typeName> : <underlyingType>
{
   <value1> = <actualVal1>,
   <value2>,
   <value3> = <value1>,
   <value4>,
   ...
   <valueN> = <actualValN>
}
```

Of course, if this is the behavior you want, then this code is fine. Note also that assigning values in a circular fashion will cause an error:

```
enum <typeName> : <underlyingType>
{
   <value1> = <value2>,
   <value2> = <value1>
}
```

The following Try It Out shows an example of all of this. The code defines and then uses an enumeration called `orientation`.

TRY IT OUT Using an Enumeration

1. Create a new console application called Ch05Ex02 and save it in the directory `C:\BegVCSharp\Chapter05`.

2. Add the following code to `Program.cs`:


```
namespace Ch05Ex02
{
    enum orientation : byte
    {
       north = 1,
       south = 2,
       east  = 3,
       west  = 4
    }

    class Program
    {
       static void Main(string[] args)
       {
          orientation myDirection = orientation.north;
          Console.WriteLine("myDirection = {0}", myDirection);
          Console.ReadKey();
       }
    }
}
```

Code snippet Ch05Ex02\Program.cs

3. Execute the application. You should see the output shown in Figure 5-6.

FIGURE 5-6

4. Quit the application and modify the code as follows:

```
byte directionByte;
string directionString;
orientation myDirection = orientation.north;
Console.WriteLine("myDirection = {0}", myDirection);
directionByte = (byte)myDirection;
directionString = Convert.ToString(myDirection);
Console.WriteLine("byte equivalent = {0}", directionByte);
Console.WriteLine("string equivalent = {0}", directionString);
Console.ReadKey();
```

5. Execute the application again. The output is shown in Figure 5-7.

FIGURE 5-7

How It Works

This code defines and uses an enumeration type called `orientation`. The first thing to notice is that the type definition code is placed in your namespace, `Ch05Ex02`, but not in the same place as the rest of your code. That is because definitions are not executed; that is, at runtime you don't step through the code in a definition as you do the lines of code in your application. Application execution starts in the place you're used to and has access to your new type because it belongs to the same namespace.

The first iteration of the example demonstrates the basic method of creating a variable of your new type, assigning it a value and outputting it to the screen. Next, you modify the code to show the conversion of enumeration values into other types. Note that you must use explicit conversions here. Even though the underlying type of `orientation` is `byte`, you still have to use the `(byte)` cast to convert the value of `myDirection` into a `byte` type:

```
directionByte = (byte)myDirection;
```

The same explicit casting is necessary in the other direction, too, if you want to convert a `byte` into an `orientation`. For example, you could use the following code to convert a `byte` variable called `myByte` into an `orientation` and assign this value to `myDirection`:

```
myDirection = (orientation)myByte;
```

Of course, care must be taken here because not all permissible values of `byte` type variables map to defined `orientation` values. The `orientation` type can store other `byte` values, so you won't get an error straight away, but this may break logic later in the application.

To get the string value of an enumeration value you can use `Convert.ToString()`:

```
directionString = Convert.ToString(myDirection);
```

Using a `(string)` cast won't work because the processing required is more complicated than just placing the data stored in the enumeration variable into a `string` variable. Alternatively, you can use the `ToString()` command of the variable itself. The following code gives you the same result as using `Convert.ToString()`:

```
directionString = myDirection.ToString();
```

Converting a `string` to an enumeration value is also possible, except that here the syntax required is slightly more complex. A special command exists for this sort of conversion, `Enum.Parse()`, which is used in the following way:

```
(enumerationType)Enum.Parse(typeof(enumerationType), enumerationValueString);
```

This uses another operator, `typeof`, which obtains the type of its operand. You could use this for your `orientation` type as follows:

```
string myString = "north";
orientation myDirection = (orientation)Enum.Parse(typeof(orientation),
                                                  myString);
```

Of course, not all string values will map to an `orientation` value! If you pass in a value that doesn't map to one of your enumeration values, you will get an error. Like everything else in C#, these values are case sensitive, so you still get an error if your string agrees with a value in everything but case (for example, if `myString` is set to `North` rather than `north`).

Structs

The *struct* (short for structure) is just that. That is, structs are data structures are composed of several pieces of data, possibly of different types. They enable you to define your own types of variables based on this structure. For example, suppose that you want to store the route to a location from a starting point, where the route consists of a direction and a distance in miles. For simplicity you can assume that the direction is one of the compass points (such that it can be represented using the `orientation` enumeration from the last section), and that distance in miles can be represented as a `double` type.

You could use two separate variables for this using code you've seen already:

```
orientation myDirection;
double      myDistance;
```

There is nothing wrong with using two variables like this, but it is far simpler (especially where multiple routes are required) to store this information in one place.

Defining Structs

Structs are defined using the struct keyword as follows:

```
struct <typeName>
{
   <memberDeclarations>
}
```

The `<memberDeclarations>` section contains declarations of variables (called the *data members* of the struct) in almost the same format as usual. Each member declaration takes the following form:

```
<accessibility> <type> <name>;
```

To allow the code that calls the struct to access the struct's data members, you use the keyword `public` for `<accessibility>`. For example:

```
struct route
{
   public orientation direction;
   public double      distance;
}
```

Once you have a struct type defined, you use it by defining variables of the new type:

```
route myRoute;
```

In addition, you have access to the data members of this composite variable via the period character:

```
myRoute.direction = orientation.north;
myRoute.distance  = 2.5;
```

This is illustrated in the following Try It Out, where the `orientation` enumeration from the last Try It Out is used with the `route` struct shown earlier. This struct is then manipulated in code to give you a feel for how structs work.

TRY IT OUT Using a Struct

1. Create a new console application called Ch05Ex03 and save it in the directory `C:\BegVCSharp\Chapter05`.

2. Add the following code to `Program.cs`:


```
namespace Ch05Ex03
{
   enum orientation: byte
   {
      north = 1,
      south = 2,
      east  = 3,
      west  = 4
   }

   struct route
   {
      public orientation direction;
      public double      distance;
   }

   class Program
   {
      static void Main(string[] args)
      {
         route myRoute;
         int myDirection = -1;
         double myDistance;
         Console.WriteLine("1) North\n2) South\n3) East\n4) West");
         do
         {
            Console.WriteLine("Select a direction:");
            myDirection = Convert.ToInt32(Console.ReadLine());
         }
         while ((myDirection < 1) || (myDirection > 4));
         Console.WriteLine("Input a distance:");
         myDistance = Convert.ToDouble(Console.ReadLine());
         myRoute.direction = (orientation)myDirection;
         myRoute.distance = myDistance;
         Console.WriteLine("myRoute specifies a direction of {0} and a " +
            "distance of {1}", myRoute.direction, myRoute.distance);
         Console.ReadKey();
      }
   }
}
```

Code snippet Ch05Ex03\Program.cs

3. Execute the code, select a direction by entering a number between 1 and 4, and then enter a distance. The result is shown in Figure 5-8.

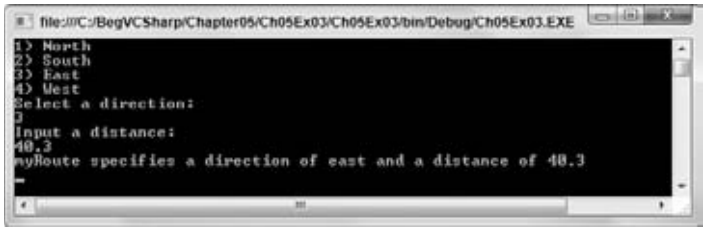

FIGURE 5-8

How It Works

Structs, like enumerations, are declared outside of the main body of the code. You declare your `route` struct just inside the namespace declaration, along with the `orientation` enumeration that it uses:

```
enum orientation: byte
{
   north = 1,
   south = 2,
   east  = 3,
   west  = 4
}

struct route
{
   public orientation direction;
   public double      distance;
}
```

The main body of the code follows a structure similar to some of the example code you've already seen, requesting input from the user and displaying it. You perform some simple validation of user input by placing the direction selection in a `do` loop, rejecting any input that isn't an integer between 1 and 4 (with values chosen such that they map onto the enumeration members for easy assignment).

NOTE *Input that cannot be interpreted as an integer will result in an error. You'll see why this happens, and what to do about it, later in the book.*

The interesting point to note is that when you refer to members of `route` they are treated exactly the same way that variables of the same type as the member would be. The assignment is as follows:

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

You could simply take the input value directly into `myRoute.distance` with no ill effects as follows:

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

The extra step allows for more validation, although none is performed in this code. Any access to members of a structure is treated in the same way. Expressions of the form `<structVar>.<memberVar>` can be said to evaluate to a variable of the type of `<memberVar>`.

Arrays

All the types you've seen so far have one thing in common: Each of them stores a single value (or a single set of values in the case of structs). Sometimes, in situations where you want to store a lot of data, this isn't very convenient. You may want to store several values of the same type at the same time, without having to use a different variable for each value.

For example, suppose you want to perform some processing that involves the names of all your friends. You could use simple string variables as follows:

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

But this looks like it will require a lot of effort, especially because you need to write different code to process each variable. You couldn't, for example, iterate through this list of strings in a loop.

The alternative is to use an *array*. Arrays are indexed lists of variables stored in a single array type variable. For example, you might have an array called `friendNames` that stores the three names shown in the preceding string variables. You can access individual members of the array by specifying their index in square brackets, as shown here:

```
friendNames[<index>]
```

The index is simply an integer, starting with 0 for the first entry, using 1 for the second, and so on. This means that you can go through the entries using a loop:

```
int i;
for (i = 0; i < 3; i++)
{
   Console.WriteLine("Name with index of {0}: {1}", i, friendNames[i]);
}
```

Arrays have a single *base type* — that is, individual entries in an array are all of the same type. This `friendNames` array has a base type of `string` because it is intended for storing `string` variables. Array entries are often referred to as *elements*.

Declaring Arrays

Arrays are declared in the following way:

```
<baseType>[] <name>;
```

Here, `<baseType>` may be any variable type, including the enumeration and struct types you've seen in this chapter. Arrays must be initialized before you have access to them. You can't just access or assign values to the array elements like this:

```
int[] myIntArray;
myIntArray[10] = 5;
```

Arrays can be initialized in two ways. You can either specify the complete contents of the array in a literal form or specify the size of the array and use the `new` keyword to initialize all array elements.

Specifying an array using literal values simply involves providing a comma-separated list of element values enclosed in curly braces:

```
int[] myIntArray = { 5, 9, 10, 2, 99 };
```

Here, `myIntArray` has five elements, each with an assigned integer value.

The other method requires the following syntax:

```
int[] myIntArray = new int[5];
```

Here, you use the `new` keyword to explicitly initialize the array, and a constant value to define the size. This method results in all the array members being assigned a default value, which is 0 for numeric types. You can also use nonconstant variables for this initialization:

```
int[] myIntArray = new int[arraySize];
```

In addition, you can combine these two methods of initialization if you want:

```
int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };
```

With this method the sizes *must* match. You can't, for example, write the following:

```
int[] myIntArray = new int[10] { 5, 9, 10, 2, 99 };
```

Here, the array is defined as having 10 members, but only five are defined, so compilation will fail. A side effect of this is that if you define the size using a variable, then that variable must be a constant:

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] { 5, 9, 10, 2, 99 };
```

If you omit the `const` keyword, this code will fail.

As with other variable types, there is no need to initialize an array on the same line that you declare it. The following is perfectly legal:

```
int[] myIntArray;
myIntArray = new int[5];
```

In the following Try It Out you create and use an array of strings, using the example from the introduction to this section.

TRY IT OUT Using an Array

1. Create a new console application called Ch05Ex04 and save it in the directory `C:\BegVCSharp\Chapter05`.
2. Add the following code to `Program.cs`:


```
static void Main(string[] args)
{
   string[] friendNames = { "Robert Barwell", "Mike Parry",
                            "Jeremy Beacock" };
   int i;
   Console.WriteLine("Here are {0} of my friends:",
                     friendNames.Length);
```

```
        for (i = 0; i < friendNames.Length; i++)
        {
            Console.WriteLine(friendNames[i]);
        }
        Console.ReadKey();
    }
```

Code snippet Ch05Ex04\Program.cs

3. Execute the code. The result is shown in Figure 5-9.

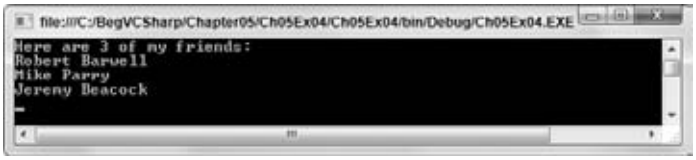

FIGURE 5-9

How It Works

This code sets up a `string` array with three values and lists them in the console in a `for` loop. Note that you have access to the number of elements in the array using `friendNames.Length`:

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

This is a handy way to get the size of an array. Outputting values in a `for` loop is easy to get wrong. For example, try changing < to <= as follows:

```
for (i = 0; i <= friendNames.Length; i++)
{
    Console.WriteLine(friendNames[i]);
}
```

Compiling the preceding code results the dialog shown in Figure 5-10.

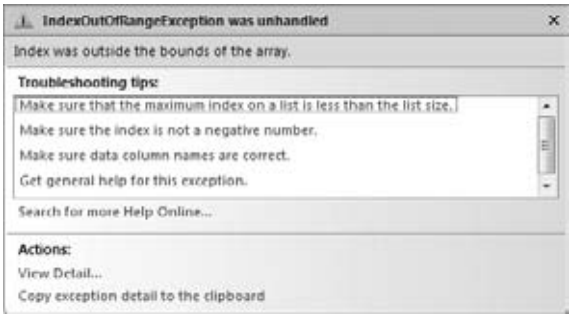

FIGURE 5-10

Here, the code attempted to access `friendNames[3]`. Remember that array indices start from 0, so the last element is `friendNames[2]`. If you attempt to access elements outside of the array size, the code will fail. It just so happens that there is a more resilient method of accessing all the members of an array: using `foreach` loops.

foreach Loops

A `foreach` loop enables you to address each element in an array using this simple syntax:

```
foreach (<baseType> <name> in <array>)
{
   // can use <name> for each element
}
```

This loop will cycle through each element, placing it in the variable *<name>* in turn, without danger of accessing illegal elements. You don't have to worry about how many elements are in the array, and you can be sure that you'll get to use each one in the loop. Using this approach, you can modify the code in the last example as follows:

```
static void Main(string[] args)
{
   string[] friendNames = { "Robert Barwell", "Mike Parry",
                            "Jeremy Beacock" };
   Console.WriteLine("Here are {0} of my friends:",
                     friendNames.Length);
   foreach (string friendName in friendNames)
   {
      Console.WriteLine(friendName);
   }
   Console.ReadKey();
}
```

The output of this code will be exactly the same as that of the previous Try It Out. The main difference between using this method and a standard `for` loop is that `foreach` gives you *read-only* access to the array contents, so you can't change the values of any of the elements. You couldn't, for example, do the following:

```
foreach (string friendName in friendNames)
{
   friendName = "Rupert the bear";
}
```

If you try this, compilation will fail. If you use a simple `for` loop, however, you can assign values to array elements.

Multidimensional Arrays

A multidimensional array is simply one that uses multiple indices to access its elements. For example, suppose you want to plot the height of a hill against the position measured. You might specify a position using two coordinates, `x` and `y`. You want to use these two coordinates as indices, such that an array called `hillHeight` would store the height at each pair of coordinates. This involves using multidimensional arrays.

A two-dimensional array such as this is declared as follows:

```
<baseType>[,] <name>;
```

Arrays of more dimensions simply require more commas:

```
<baseType>[,,,] <name>;
```

This would declare a four-dimensional array. Assigning values also uses a similar syntax, with commas separating sizes. Declaring and initializing the two-dimensional array `hillHeight`, with a base type of `double`, an `x` size of 3, and a `y` size of 4 requires the following:

```
double[,] hillHeight = new double[3,4];
```

Alternatively, you can use literal values for initial assignment. Here, you use nested blocks of curly braces, separated by commas:

```
double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };
```

This array has the same dimensions as the previous one — that is, three rows and four columns. By providing literal values, these dimensions are defined implicitly.

To access individual elements of a multidimensional array, you simply specify the indices separated by commas:

```
hillHeight[2,1]
```

You can then manipulate this element just as you can other elements. This expression will access the second element of the third nested array as defined previously (the value will be 4). Remember that you start counting from 0 and that the first number is the nested array. In other words, the first number specifies the pair of curly braces, and the second number specifies the element within that pair of braces. You can represent this array visually, as shown in Figure 5-11.

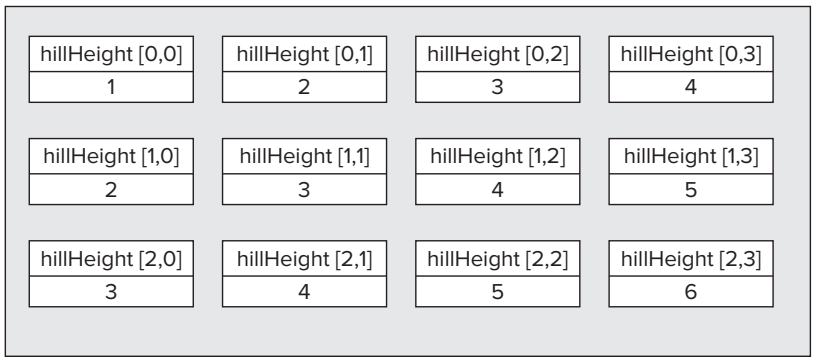

FIGURE 5-11

The `foreach` loop gives you access to all elements in a multidimensional way, just as with single-dimensional arrays:

```
double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };
foreach (double height in hillHeight)
{
   Console.WriteLine("{0}", height);
}
```

The order in which the elements are output is the same as the order used to assign literal values. This sequence is as follows (the element identifiers are shown here rather than the actual values):

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
...
```

Arrays of Arrays

Multidimensional arrays, as discussed in the last section, are said to be *rectangular* because each "row" is the same size. Using the last example, you can have a `y` coordinate of 0 to 3 for any of the possible `x` coordinates.

It is also possible to have *jagged* arrays, whereby "rows" may be different sizes. For this, you need an array in which each element is another array. You could also have arrays of arrays of arrays, or even more complex situations. However, all this is possible only if the arrays have the same base type.

The syntax for declaring arrays of arrays involves specifying multiple sets of square brackets in the declaration of the array, as shown here:

```
int[][] jaggedIntArray;
```

Unfortunately, initializing arrays such as this isn't as simple as initializing multidimensional arrays. You can't, for example, follow the preceding declaration with this:

```
jaggedIntArray = new int[3][4];
```

Even if you could do this, it wouldn't be that useful because you can achieve the same effect with simple multidimensional arrays with less effort. Nor can you use code such as this:

```
jaggedIntArray = { { 1, 2, 3 }, { 1 }, { 1, 2 } };
```

You have two options. You can initialize the array that contains other arrays (we'll call these sub-arrays for clarity) and then initialize the sub-arrays in turn:

```
jaggedIntArray = new int[2][];
jaggedIntArray[0] = new int[3];
jaggedIntArray[1] = new int[4];
```

Alternately, you can use a modified form of the preceding literal assignment:

```
jaggedIntArray = new int[3][] { new int[] { 1, 2, 3 }, new int[] { 1 },
                                new int[] { 1, 2 } };
```

This can be simplified if the array is initialized on the same line as it is declared, as follows:

```
int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] { 1 },
                           new int[] { 1, 2 } };
```

You can use `foreach` loops with jagged arrays, but you often need to nest these to get to the actual data. For example, suppose you have the following jagged array that contains 10 arrays, each of which contains an array of integers that are divisors of an integer between 1 and 10:

```
int[][] divisors1To10 = { new int[] { 1 },
                          new int[] { 1, 2 },
                          new int[] { 1, 3 },
                          new int[] { 1, 2, 4 },
                          new int[] { 1, 5 },
                          new int[] { 1, 2, 3, 6 },
                          new int[] { 1, 7 },
                          new int[] { 1, 2, 4, 8 },
                          new int[] { 1, 3, 9 },
                          new int[] { 1, 2, 5, 10 } };
```

The following code will fail:

```
foreach (int divisor in divisors1To10)
{
   Console.WriteLine(divisor);
}
```

That's because the array `divisors1To10` contains `int[]` elements, not `int` elements. Instead, you have to loop through every sub-array as well as through the array itself:

```
foreach (int[] divisorsOfInt in divisors1To10)
{
   foreach(int divisor in divisorsOfInt)
   {
      Console.WriteLine(divisor);
   }
}
```

As you can see, the syntax for using jagged arrays can quickly become complex! In most cases, it is easier to use rectangular arrays or a simpler storage method. Nonetheless, there may well be situations in which you are forced to use this method, and a working knowledge can't hurt.

STRING MANIPULATION

Your use of strings so far has consisted of writing strings to the console, reading strings from the console, and concatenating strings using the + operator. In the course of programming more interesting applications, you will discover that manipulating strings is something that you end up doing *a lot*. Therefore, it is worth spending a few pages looking at some of the more common string manipulation techniques available in C#.

To start with, a `string` type variable can be treated as a read-only array of `char` variables. This means that you can access individual characters using syntax like the following:

```
string myString = "A string";
char myChar = myString[1];
```

However, you can't assign individual characters this way. To get a `char` array that you can write to, you can use the following code. This uses the `ToCharArray()` command of the array variable:

```
string myString = "A string";
char[] myChars = myString.ToCharArray();
```

Then you can manipulate the `char` array the standard way. You can also use strings in `foreach` loops, as shown here:

```
foreach (char character in myString)
{
   Console.WriteLine("{0}", character);
}
```

As with arrays, you can also get the number of elements using `myString.Length`. This gives you the number of characters in the string:

```
string myString = Console.ReadLine();
Console.WriteLine("You typed {0} characters.", myString.Length);
```

Other basic string manipulation techniques use commands with a format similar to this `<string>.ToCharArray()` command. Two simple, but useful, ones are `<string>.ToLower()` and `<string>.ToUpper()`. These enable strings to be converted into lowercase and uppercase, respectively. To see why this is useful, consider the situation in which you want to check for a specific response from a user — for example, the string `yes`. If you convert the string entered by the user into lowercase, then you can also check for the strings `YES`, `Yes`, `yeS`, and so on — you saw an example of this in the previous chapter:

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
   // Act on response.
}
```

This command, like the others in this section, doesn't actually change the string to which it is applied. Instead, combining this command with a string results in the creation of a new string, which you can compare to another string (as shown here) or assign to another variable. The other variable may be the same one that is being operated on:

```
userResponse = userResponse.ToLower();
```

This is an important point to remember, because just writing

```
userResponse.ToLower();
```

doesn't actually achieve very much!

There are other things you can do to ease the interpretation of user input. What if the user accidentally put an extra space at the beginning or end of the input? In this case, the preceding code won't work. You need to trim the string entered, which you can do using the `<string>.Trim()` command:

```
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
   // Act on response.
}
```

The preceding code is also able detect strings like this:

```
" YES"
"Yes "
```

You can also use these commands to remove any other characters, by specifying them in a `char` array, for example:

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
   // Act on response.
}
```

This eliminates any occurrences of spaces, the letter "`e`," and the letter "`s`" from the beginning or end of your string. Providing there aren't any other characters in the string, this will result in the detection of strings such as

```
"Yeeeees"
" y"
```

and so on.

You can also use the `<string>.TrimStart()` and `<string>.TrimEnd()` commands, which will trim spaces from the beginning and end of a string, respectively. These can also have `char` arrays specified.

You can use two other string commands to manipulate the spacing of strings: `<string>.PadLeft()` and `<string>.PadRight()`. They enable you to add spaces to the left or right of a string to force it to the desired length. You use them as follows:

```
<string>.PadX(<desiredLength>);
```

Here is an example:

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

This would result in three spaces being added to the left of the word `Aligned` in `myString`. These methods can be helpful when aligning strings in columns, which is particularly useful for positioning strings containing numbers.

As with the trimming commands, you can also use these commands in a second way, by supplying the character to pad the string with. This involves a single `char`, not an array of `char`s as with trimming:

```
myString = "Aligned";
myString = myString.PadLeft(10, '-');
```

This would add three dashes to the start of `myString`.

There are many more of these string manipulation commands, many of which are only useful in very specific situations. These are discussed as you use them in the forthcoming chapters. Before moving on, though, it is worth looking at one of the features contained in both Visual C# 2010 Express Edition and Visual Studio 2010 that you may have noticed over the course of the last few chapters, and especially this one. In the following Try It Out, you examine auto-completion, whereby the IDE tries to help you out by suggesting what code you might like to insert.

TRY IT OUT Statement Auto-Completion in VS

1. Create a new console application called Ch05Ex05 and save it in the directory `C:\BegVCSharp\Chapter05`.

2. Type the following code into `Program.cs`, exactly as written, noting windows that pop up as you do so:


```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.
}
```

Code snippet Ch05Ex05\Program.cs

FIGURE 5-12

3. As you type the final period, the window shown in Figure 5-12 appears.

4. Without moving the cursor, type **sp**. The pop-up window changes, and the tooltip shown in Figure 5-13 appears (it is yellow, which can't be seen in the screenshot).

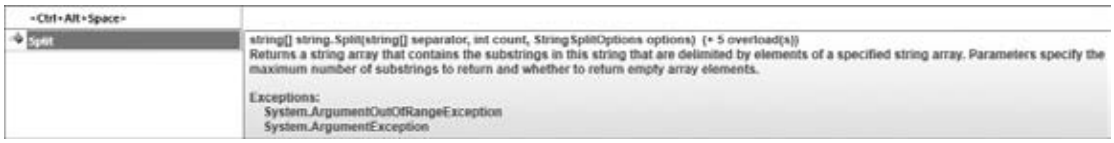

FIGURE 5-13

5. Type the following characters: **`(se`**. Another pop-up window appears, as shown in Figure 5-14.

6. Then type these two characters: **`);`**. The code should look as follows, and the pop-up windows should disappear:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
}
```

FIGURE 5-14

7. Add the following code, noting the windows as they pop up:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
   foreach (string word in myWords)
   {
      Console.WriteLine("{0}", word);
   }
   Console.ReadKey();
}
```

8. Execute the code. The result is shown in Figure 5-15.

FIGURE 5-15

How It Works

Two main aspects of this code are the new string command used and the use of the auto-completion functionality. The command, `<string>.Split()`, converts a `string` into a `string` array by splitting it at the points specified. These points take the form of a `char` array, which in this case is simply populated by a single element, the space character:

```
char[] separator = {' '};
```

The following code obtains the substrings you get when the string is split at each space — that is, you get an array of individual words:

```
string[] myWords;
myWords = myString.Split(separator);
```

Next, you loop through the words in this array using `foreach` and write each one to the console:

```
foreach (string word in myWords)
{
   Console.WriteLine("{0}", word);
}
```

NOTE *Each word obtained has no spaces, either embedded in the word or at either end. The separators are removed when you use* `Split()`.

Next, on to auto-completion. Both VS and VCE are very intelligent packages that work out a lot of information about your code as you type it in. Even as you type the first character on a new line, the IDE tries to help you by suggesting a keyword, a variable name, a type name, and so on. Only three letters into the preceding code (`str`), the IDE correctly guessed that you want to type `string`. Even more useful is when you type variable names. In long pieces of code, you often forget the names of variables you want to use. Because the IDE pops up a list of these as you type, you can find them easily, without having to refer to earlier code.

By the time you type the period after `myString`, it knows that `myString` is a string, detects that you want to specify a string command, and presents the available options. At this point, you can stop typing if desired, and select the command you want using the up and down arrow keys. As you move through the available options, the IDE describes the currently selected command and indicates what syntax it uses.

As you start typing more characters, the IDE moves the selected command to the top of the commands you might mean automatically. Once it shows the command you want, you can simply carry on typing as if you'd typed the whole name, so typing "`(`" takes you straight to the point where you specify the additional information that some commands require — and the IDE even tells you the format this extra information must be in, presenting options for those commands that accept varying amounts of information.

This feature of the IDE, known as IntelliSense, comes in very handy, enabling you to find information about strange types with ease. You might find it interesting to look at all the commands that the `string` type exposes and experiment — nothing you do is going to break the computer, so play away!

NOTE *Sometimes the displayed information can obscure some of the code you have already typed, which can be annoying. This is because the hidden code may be something that you need to refer to when typing. However, you can press the Ctrl key to make the command list transparent, enabling you to see what was hidden.*

SUMMARY

In this chapter, you've spent some time expanding your knowledge of variables. Perhaps the most important topic covered in this chapter is type conversion, which will appear again throughout this book. Getting a sound grasp of the concepts involved now will make things a lot easier later.

You've also seen a few more variable types that you can use to help you store data in a more developer-friendly way. You've learned how enumerations can make your code much more readable with easily discernable values; how structs can be used to combine multiple, related data elements in one place; and how you can group similar data together in arrays. You see all of these types used many times throughout the rest of this book.

Finally, you looked at string manipulation, including some of the basic techniques and principles involved. Many individual string commands are available, and although you only examined a few,

you now know how to view the available commands in your IDE. Using this technique, you can have some fun trying things out. At least one of the following exercises can be solved using one or more string commands you haven't seen yet, but you'll have to figure out which!

This chapter extended your knowledge of variables to cover the following:

- Type conversions
- Enumerations
- Structs
- Arrays
- String manipulation

EXERCISES

1. Which of the following conversions can't be performed implicitly?

a. `int` to `short`
b. `short` to `int`
c. `bool` to `string`
d. `byte` to `float`

2. Show the code for a `color` enumeration based on the `short` type containing the colors of the rainbow plus black and white. Can this enumeration be based on the `byte` type?

3. Modify the Mandelbrot set generator example from the last chapter to use the following struct for complex numbers:

```
struct imagNum
{
   public double real, imag;
}
```

4. Will the following code compile? Why or why not?

```
string[] blab = new string[5]
string[5] = 5th string.
```

5. Write a console application that accepts a string from the user and outputs a string with the characters in reverse order.

6. Write a console application that accepts a string and replaces all occurrences of the string `no` with `yes`.

7. Write a console application that places double quotes around each word in a string.

Answers to Exercises can be found in Appendix A.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPT
Type conversion	Values can be converted from one type into another, but there are rules that apply when you do so. Implicit conversion happens automatically, but only when all possible values of the source value type are available in the target value type. Explicit conversion is also possible, but you run the risk of values not being assigned as expected, or even causing errors.
Enumerations	Enums, or enumerations, are types that have a discrete set of values, each of which has a name. Enums are defined with the `enum` keyword. This makes them easy to understand in code because they are very readable. Enums have an underlying numeric type (`int` by default), and you can use this property of enum values to convert between enum values and numeric values, or to identify enum values.
Structs	Structs, or structures, are types that contain several different values at the same time. Structs are defined with the `struct` keyword. The values contained in a struct each have a name and a type; there is no requirement that every value stored in a struct is the same type.
Arrays	An array is a collection of values of the same type. Arrays have a fixed size, or length, which determines how many values they can contain. You can define multidimensional or jagged arrays to hold different amounts and shapes of data. You can also iterate through the values in an array with a `foreach` loop.

6

Functions

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to define and use simple functions that don't accept or return any data
- How to transfer data to and from functions
- Working with variable scope
- How to use command-line arguments with the `Main()` function
- How to supply functions as members of struct types
- How to use function overloading
- How to use delegates

All the code you have seen so far has taken the form of a single block, perhaps with some looping to repeat lines of code, and branching to execute statements conditionally. Performing an operation on your data has meant placing the code required right where you want it to work.

This kind of code structure is limited. Often, some tasks — such as finding the highest value in an array, for example — may need to be performed at several points in a program. You can place identical (or nearly identical) sections of code in your application whenever necessary, but this has its own problems. Changing even one minor detail concerning a common task (to correct a code error, for example) may require changes to multiple sections of code, which may be spread throughout the application. Missing one of these could have dramatic consequences and cause the whole application to fail. In addition, the application could get very lengthy.

The solution to this problem is to use *functions*. Functions in C# are a means of providing blocks of code that can be executed at any point in an application.

NOTE *Functions of the specific type examined in this chapter are known as methods, but this term has a very specific meaning in .NET programming that will only become clear later in this book. Therefore, for now, the term method will not be used.*

For example, you could have a function that calculates the maximum value in an array. You can use the function from any point in your code, and use the same lines of code in each case. Because you only need to supply this code once, any changes you make to it will affect this calculation wherever it is used. The function can be thought of as containing *reusable* code.

Functions also have the advantage of making your code more readable, as you can use them to group related code together. This way, your application body itself can be made very short, as the inner workings of the code are separated out. This is similar to the way in which you can collapse regions of code together in the IDE using the outline view, and it gives your application a more logical structure.

Functions can also be used to create multipurpose code, enabling them to perform the same operations on varying data. You can supply a function with information to work with in the form of parameters, and you can obtain results from functions in the form of return values. In the preceding example, you could supply an array to search as a parameter and obtain the maximum value in the array as a return value. This means that you can use the same function to work with a different array each time. The name and parameters of a function (but not its return type) collectively define the *signature* of a function.

DEFINING AND USING FUNCTIONS

This section describes how you can add functions to your applications and then use (call) them from your code. Starting with the basics, you look at simple functions that don't exchange any data with code that calls them, and then look at more advanced function usage. The following Try It Out gets things moving.

TRY IT OUT Defining and Using a Basic Function

1. Create a new console application called Ch06Ex01 and save it in the directory `C:\BegVCSharp\Chapter06`.
2. Add the following code to `Program.cs`:


```
class Program
{
    static void Write()
    {
        Console.WriteLine("Text output from function.");
    }
```

```
    static void Main(string[] args)
    {
        Write();
        Console.ReadKey();
    }
}
```

Code snippet Ch06Ex01\Program.cs

3. Execute the code. The result is shown in Figure 6-1.

FIGURE 6-1

How It Works

The following four lines of your code define a function called `Write()`:

```
static void Write()
{
    Console.WriteLine("Text output from function.");
}
```

The code contained here simply outputs some text to the console window, but this behavior isn't that important at the moment, because the focus here is on the mechanisms behind function definition and use.

The function definition consists of the following:

- Two keywords: `static` and `void`
- A function name followed by parentheses: `Write()`
- A block of code to execute, enclosed in curly braces

NOTE *Function names are usually written in PascalCase.*

The code that defines the `Write()` function looks very similar to some of the other code in your application:

```
static void Main(string[] args)
{
    ...
}
```

That's because all the code you have written so far (apart from type definitions) has been part of a function. This function, `Main()`, is the *entry point* function for a console application. When a C# application is executed, the entry point function it contains is called; and when that function is completed, the application terminates. All C# executable code must have an entry point.

The only difference between the `Main()` function and your `Write()` function (apart from the lines of code they contain) is that there is some code inside the parentheses after the function name `Main`. This is how you specify parameters, which you see in more detail shortly.

As mentioned earlier, both `Main()` and `Write()` are defined using the `static` and `void` keywords. The `static` keyword relates to object-oriented concepts, which you come back to later in the book. For now, you only need to remember that all the functions you use in your applications in this section of the book must use this keyword.

In contrast, `void` is much simpler to explain. It's used to indicate that the function does not return a value. Later in this chapter, you'll see the code that you need to use when a function has a return value.

Moving on, the code that calls your function is as follows:

```
Write();
```

You simply type the name of the function followed by empty parentheses. When program execution reaches this point, the code in the `Write()` function runs.

NOTE *The parentheses used both in the function definition and where the function is called are mandatory. Try removing them if you like — the code won't compile.*

Return Values

The simplest way to exchange data with a function is to use a return value. Functions that have return values *evaluate* to that value exactly the same way that variables evaluate to the values they contain when you use them in expressions. Just like variables, return values have a type.

For example, you might have a function called `GetString()` whose return value is a string. You could use this in code, such as the following:

```
string myString;
myString = GetString();
```

Alternatively, you might have a function called `GetVal()` that returns a `double` value, which you could use in a mathematical expression:

```
double myVal;
double multiplier = 5.3;
myVal = GetVal() * multiplier;
```

When a function returns a value, you have to modify your function in two ways:

- Specify the type of the return value in the function declaration instead of using the `void` keyword.
- Use the `return` keyword to end the function execution and transfer the return value to the calling code.

In code terms, this looks like the following in a console application function of the type you've been looking at:

```
static <returnType> <FunctionName>()
{
   ...
   return <returnValue>;
}
```

The only limitation here is that *<returnValue>* must be a value that either is of type *<returnType>* or can be implicitly converted to that type. However, *<returnType>* can be any type you want, including the more complicated types you've seen. This might be as simple as the following:

```
static double GetVal()
{
   return 3.2;
}
```

However, return values are usually the result of some processing carried out by the function; the preceding could be achieved just as easily using a `const` variable.

When the `return` statement is reached, program execution returns to the calling code immediately. No lines of code after this statement are executed, although this doesn't mean that `return` statements can only be placed on the last line of a function body. You can use `return` earlier in the code, perhaps after performing some branching logic. Placing `return` in a `for` loop, an `if` block, or any other structure causes the structure to terminate immediately and the function to terminate:

```
static double GetVal()
{
   double checkVal;
   // CheckVal assigned a value through some logic (not shown here).
   if (checkVal < 5)
      return 4.7;
      return 3.2;
}
```

Here, one of two values may be returned, depending on the value of `checkVal`. The only restriction in this case is that a `return` statement must be processed before reaching the closing `}` of the function. The following is illegal:

```
static double GetVal()
{
   double checkVal;
   // CheckVal assigned a value through some logic.
   if (checkVal < 5)
      return 4.7;
}
```

If `checkVal` is `>= 5`, then no `return` statement is met, which isn't allowed. All processing paths must reach a `return` statement. In most cases, the compiler detects this and gives you the error "not all code paths return a value."

As a final note, `return` can be used in functions that are declared using the `void` keyword (those that don't have a return value). In that case, the function simply terminates. When you use `return` this

way, it is an error to provide a return value between the `return` keyword and the semicolon that follows.

Parameters

When a function is to accept parameters, you must specify the following:

- A list of the parameters accepted by the function in its definition, along with the types of those parameters
- A matching list of parameters in each function call

This involves the following code, where you can have any number of parameters, each with a type and a name:

```
static <returnType> <FunctionName>(<paramType> <paramName>, ...)
{
   ...
   return <returnValue>;
}
```

The parameters are separated using commas, and each of these parameters is accessible from code within the function as a variable. For example, a simple function might take two `double` parameters and return their product:

```
static double Product(double param1, double param2)
{
   return param1 * param2;
}
```

The following Try It Out provides a more complex example.

TRY IT OUT Exchanging Data with a Function (Part 1)

1. Create a new console application called Ch06Ex02 and save it in the directory `C:\BegVCSharp\Chapter06`.
2. Add the following code to `Program.cs`:


```
class Program
{
   static int MaxValue(int[] intArray)
   {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
         if (intArray[i] > maxVal)
            maxVal = intArray[i];
      }
      return maxVal;
   }

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

```
            int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
            int maxVal = MaxValue(myArray);
            Console.WriteLine("The maximum value in myArray is {0}", maxVal);
            Console.ReadKey();
        }
    }
```

Code snippet Ch06Ex02\Program.cs

3. Execute the code. The result is shown in Figure 6-2.

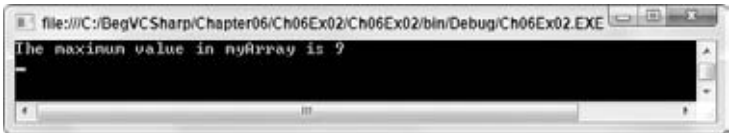

FIGURE 6-2

How It Works

This code contains a function that does what the example function at the beginning of this chapter hoped to do. It accepts an array of integers as a parameter and returns the highest number in the array. The function definition is as follows:

```
static int MaxValue(int[] intArray)
{
   int maxVal = intArray[0];
   for (int i = 1; i < intArray.Length; i++)
   {
      if (intArray[i] > maxVal)
      maxVal = intArray[i];
   }
   return maxVal;
}
```

The function, `MaxValue()`, has a single parameter defined, an `int` array called `intArray`. It also has a return type of `int`. The calculation of the maximum value is simple. A local integer variable called `maxVal` is initialized to the first value in the array, and then this value is compared with each of the subsequent elements in the array. If an element contains a higher value than `maxVal`, then this value replaces the current value of `maxVal`. When the loop finishes, `maxVal` contains the highest value in the array, and is returned using the `return` statement.

The code in `Main()` declares and initializes a simple integer array to use with the `MaxValue()` function:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
```

The call to `MaxValue()` is used to assign a value to the `int` variable `maxVal`:

```
int maxVal = MaxValue(myArray);
```

Next, you write that value to the screen using `Console.WriteLine()`:

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

Parameter Matching

When you call a function, you must match the parameters as specified in the function definition exactly. This means matching the parameter types, the number of parameters, and the order of the parameters. For example, the function

```
static void MyFunction(string myString, double myDouble)
{
     ...
}
```

can't be called using the following:

```
MyFunction(2.6, "Hello");
```

Here, you are attempting to pass a `double` value as the first parameter, and a `string` value as the second parameter, which is not the order in which the parameters are defined in the function definition.

You also can't use

```
MyFunction("Hello");
```

because you are only passing a single `string` parameter, where two parameters are required. Attempting to use either of the two preceding function calls will result in a compiler error, because the compiler forces you to match the signatures of the functions you use.

NOTE *Recall from the introduction that the signature of a function is defined by the name and parameters of the function.*

Going back to the example, `MaxValue()` can only be used to obtain the maximum `int` in an array of `int` values. If you replace the code in `Main()` with

```
static void Main(string[] args)
{
     double[] myArray = { 1.3, 8.9, 3.3, 6.5, 2.7, 5.3 };
     double maxVal = MaxValue(myArray);
     Console.WriteLine("The maximum value in myArray is {0}", maxVal);
     Console.ReadKey();
}
```

the code won't compile because the parameter type is wrong. In the "Overloading Functions" section later in this chapter, you'll learn a useful technique for getting around this problem.

Parameter Arrays

C# enables you to specify one (and only one) special parameter for a function. This parameter, which must be the last parameter in the function definition, is known as a *parameter array*. Parameter arrays enable you to call functions using a variable amount of parameters, and they are defined using the `params` keyword.

Parameter arrays can be a useful way to simplify your code because you don't have to pass arrays from your calling code. Instead, you pass several parameters of the same type, which are placed in an array you can use from within your function.

The following code is required to define a function that uses a parameter array:

```
static <returnType> <FunctionName>(<p1Type> <p1Name>, ...,
                                   params <type>[] <name>)
{
    ...
    return <returnValue>;
}
```

You can call this function using code like the following:

```
<FunctionName>(<p1>, ..., <val1>, <val2>, ...)
```

<val1>, <val2>, and so on are values of type <type>, which are used to initialize the <name> array. The number of parameters that you can specify here is almost limitless; the only restriction is that they must all be of type <type>. You can even specify no parameters at all.

This final point makes parameter arrays particularly useful for specifying additional information for functions to use in their processing. For example, suppose you have a function called `GetWord()` that takes a `string` value as its first parameter and returns the first word in the string:

```
string firstWord = GetWord("This is a sentence.");
```

Here, `firstWord` will be assigned the string `This`.

You might add a `params` parameter to `GetWord()`, enabling you to optionally select an alternative word to return by its index:

```
string firstWord = GetWord("This is a sentence.", 2);
```

Assuming that you start counting at 1 for the first word, this would result in `firstWord` being assigned the string `is`.

You might also add the capability to limit the number of characters returned in a third parameter, also accessible through the `params` parameter:

```
string firstWord = GetWord("This is a sentence.", 4, 3);
```

Here, `firstWord` would be assigned the string `sen`.

The following Try It Out defines and uses a function with a `params` type parameter.

TRY IT OUT Exchanging Data with a Function (Part 2)

1. Create a new console application called Ch06Ex03 and save it in the directory `C:\BegVCSharp\Chapter06`.
2. Add the following code to `Program.cs`:


```
class Program
{
   static int SumVals(params int[] vals)
   {
      int sum = 0;
      foreach (int val in vals)
      {
         sum += val;
```

```
        }
        return sum;
    }

    static void Main(string[] args)
    {
        int sum = SumVals(1, 5, 2, 9, 8);
        Console.WriteLine("Summed Values = {0}", sum);
        Console.ReadKey();
    }
}
```

Code snippet Ch06Ex03\Program.cs

3. Execute the code. The result is shown in Figure 6-3.

FIGURE 6-3

How It Works

The function `SumVals()` is defined using the `params` keyword to accept any number of `int` parameters (and no others):

```
static int SumVals(params int[] vals)
{
   ...
}
```

The code in this function simply iterates through the values in the `vals` array and adds the values together, returning the result.

In `Main()`, you call `SumVals()` with five integer parameters:

```
int sum = SumVals(1, 5, 2, 9, 8);
```

You could just as easily call this function with none, one, two, or 100 integer parameters — there is no limit to the number you can specify.

Reference and Value Parameters

All the functions defined so far in this chapter have had value parameters. That is, when you have used parameters, you have passed a value into a variable used by the function. Any changes made to this variable in the function have no effect on the parameter specified in the function call. For example, consider a function that doubles and displays the value of a passed parameter:

```
static void ShowDouble(int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

Here, the parameter, `val`, is doubled in this function. If you call it like this,

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

then the text output to the console is as follows:

```
myNumber = 5
val doubled = 10
myNumber = 5
```

Calling `ShowDouble()` with `myNumber` as a parameter doesn't affect the value of `myNumber` in `Main()`, even though the parameter it is assigned to, `val`, is doubled.

That's all very well, but if you *want* the value of `myNumber` to change, you have a problem. You could use a function that returns a new value for `myNumber`, like this:

```
static int DoubleNum(int val)
{
   val *= 2;
   return val;
}
```

You could call this function using the following:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = DoubleNum(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

However, this code is hardly intuitive and won't cope with changing the values of multiple variables used as parameters (as functions have only one return value).

Instead, you want to pass the parameter by *reference*, which means that the function will work with exactly the same variable as the one used in the function call, not just a variable that has the same value. Any changes made to this variable will, therefore, be reflected in the value of the variable used as a parameter. To do this, you simply use the `ref` keyword to specify the parameter:

```
static void ShowDouble(ref int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

Then, specify it again in the function call (this is mandatory):

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

The text output to the console is now as follows:

```
myNumber = 5
val doubled = 10
myNumber = 10
```

This time `myNumber` has been modified by `ShowDouble()`.

Note two limitations on the variable used as a `ref` parameter. First, the function may result in a change to the value of a reference parameter, so you must use a *nonconstant* variable in the function call. The following is therefore illegal:

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Second, you must use an initialized variable. C# doesn't allow you to assume that a `ref` parameter will be initialized in the function that uses it. The following code is also illegal:

```
int myNumber;
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Out Parameters

In addition to passing values by reference, you can specify that a given parameter is an *out* parameter by using the `out` keyword, which is used in the same way as the `ref` keyword (as a modifier to the parameter in the function definition and in the function call). In effect, this gives you almost exactly the same behavior as a reference parameter, in that the value of the parameter at the end of the function execution is returned to the variable used in the function call. However, there are important differences:

- Whereas it is illegal to use an unassigned variable as a `ref` parameter, you can use an unassigned variable as an `out` parameter.
- An `out` parameter must be treated as an unassigned value by the function that uses it.

This means that while it is permissible in calling code to use an assigned variable as an `out` parameter, the value stored in this variable is lost when the function executes.

As an example, consider an extension to the `MaxValue()` function shown earlier, which returns the maximum value of an array. Modify the function slightly so that you obtain the index of the element with the maximum value within the array. To keep things simple, obtain just the index of the first occurrence of this value when there are multiple elements with the maximum value. To do this, you add an `out` parameter by modifying the function as follows:

```
static int MaxValue(int[] intArray, out int maxIndex)
{
   int maxVal = intArray[0];
   maxIndex = 0;
   for (int i = 1; i < intArray.Length; i++)
   {
      if (intArray[i] > maxVal)
      {
         maxVal = intArray[i];
         maxIndex = i;
      }
   }
   return maxVal;
}
```

You might use the function like this:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
        MaxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
        maxIndex + 1);
```

That results in the following:

```
The maximum value in myArray is 9
The first occurrence of this value is at element 7
```

You must use the `out` keyword in the function call, just as with the `ref` keyword.

NOTE *One has been added to the value of* `maxIndex` *returned here when it is displayed onscreen. This is to translate the index to a more readable form so that the first element in the array is referred to as element 1, rather than element 0.*

VARIABLE SCOPE

Throughout the last section, you may have been wondering why exchanging data with functions is necessary. The reason is that variables in C# are accessible only from localized regions of code. A given variable is said to have a *scope* from which it is accessible.

Variable scope is an important subject and one best introduced with an example. The following Try It Out illustrates a situation in which a variable is defined in one scope, and an attempt to use it is made in a different scope.

TRY IT OUT Variable Scope

1. Make the following changes to Ch06Ex01 in `Program.cs`:


```
class Program
{
   static void Write()
   {
      Console.WriteLine("myString = {0}", myString);
   }

   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Write();
      Console.ReadKey();
   }
}
```

Code snippet Ch06Ex01\Program.cs

2. Compile the code and note the error and warning that appear in the task list:

```
The name 'myString' does not exist in the current context
The variable 'myString' is assigned but its value is never used
```

How It Works

What went wrong? Well, the variable `myString` defined in the main body of your application (the `Main()` function) isn't accessible from the `Write()` function.

The reason for this inaccessibility is that variables have a scope within which they are valid. This scope encompasses the code block that they are defined in and any directly nested code blocks. The blocks of code in functions are separate from the blocks of code from which they are called. Inside `Write()`, the name `myString` is undefined, and the `myString` variable defined in `Main()` is *out of scope* — it can be used only from within `Main()`.

In fact, you can have a completely separate variable in `Write()` called `myString`. Try modifying the code as follows:

```
class Program
{
   static void Write()
   {
      string myString = "String defined in Write()";
      Console.WriteLine("Now in Write()");
      Console.WriteLine("myString = {0}", myString);
   }

   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Write();
      Console.WriteLine("\nNow in Main()");
      Console.WriteLine("myString = {0}", myString);
      Console.ReadKey();
   }
}
```

This code does compile, resulting in the output shown in Figure 6-4.

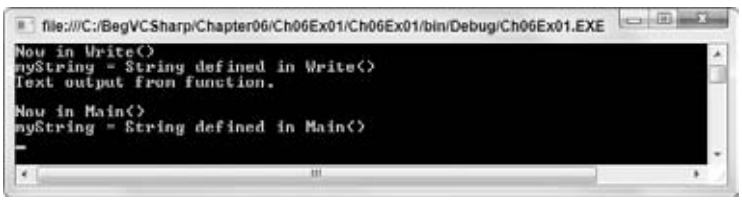

FIGURE 6-4

The operations performed by this code are as follows:

- `Main()` defines and initializes a string variable called `myString`.
- `Main()` transfers control to `Write()`.

- `Write()` defines and initializes a string variable called `myString`, which is a different variable from the `myString` defined in `Main()`.
- `Write()` outputs a string to the console containing the value of `myString` as defined in `Write()`.
- `Write()` transfers control back to `Main()`.
- `Main()` outputs a string to the console containing the value of `myString` as defined in `Main()`.

Variables whose scopes cover a single function in this way are known as *local variables*. It is also possible to have *global variables*, whose scopes cover multiple functions. Modify the code as follows:

```
class Program
{
   static string myString;

   static void Write()
   {
      string myString = "String defined in Write()";
      Console.WriteLine("Now in Write()");
      Console.WriteLine("Local myString = {0}", myString);
      Console.WriteLine("Global myString = {0}", Program.myString);
   }

   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Program.myString = "Global string";
      Write();
      Console.WriteLine("\nNow in Main()");
      Console.WriteLine("Local myString = {0}", myString);
      Console.WriteLine("Global myString = {0}", Program.myString);
      Console.ReadKey();
   }
}
```

The result is now as shown in Figure 6-5.

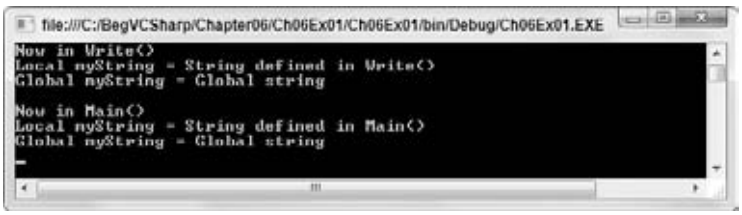

FIGURE 6-5

Here, you have added another variable called `myString`, this time further up the hierarchy of names in the code. The variable is defined as follows:

```
static string myString;
```

Again, the `static` keyword is required. Without going into too much detail, understand that in this type of console application, you must use either the `static` or the `const` keyword for global variables of this form.

If you want to modify the value of the global variable, you need to use `static` because `const` prohibits the value of the variable from changing.

To differentiate between this variable and the local variables in `Main()` and `Write()` with the same names, you have to classify the variable name using a fully qualified name, as described in Chapter 3. Here, you refer to the global version as `Program.myString`. This is only necessary when you have global and local variables with the same name; if there were no local `myString` variable, you could simply use `myString` to refer to the global variable, rather than `Program.myString`. When you have a local variable with the same name as a global variable, the global variable is said to be *hidden*.

The value of the global variable is set in `Main()` with

```
Program.myString = "Global string";
```

and accessed in `Write()` with

```
Console.WriteLine("Global myString = {0}", Program.myString);
```

You might be wondering why you shouldn't just use this technique to exchange data with functions, rather than the parameter passing shown earlier. There are indeed situations where this is the preferable way to exchange data, but there are just as many scenarios (if not more) where it isn't. The choice of whether to use global variables depends on the intended use of the function in question. The problem with using global variables is that they are generally unsuitable for "general-purpose" functions, which are capable of working with whatever data you supply, not just data in a specific global variable. You look at this in more depth a little later.

Variable Scope in Other Structures

One of the points made in the last section has consequences above and beyond variable scope between functions: that the scopes of variables encompass the code blocks in which they are defined and any directly nested code blocks. This also applies to other code blocks, such as those in branching and looping structures. Consider the following code:

```
int i;
for (i = 0; i < 10; i++)
{
   string text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

Here, the string variable `text` is local to the `for` loop. This code won't compile because the call to `Console.WriteLine()` that occurs outside of this loop attempts to use the variable `text`, which is out of scope outside of the loop. Try modifying the code as follows:

```
int i;
string text;
for (i = 0; i < 10; i++)
{
   text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

This code will also fail because variables must be declared and initialized before use, and `text` is only initialized in the `for` loop. The value assigned to `text` is lost when the loop block is exited. However, you can make the following change:


```
int i;
string text = "";
for (i = 0; i < 10; i++)
{
   text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

Code snippet VariableScopeInLoops\Program.cs

This time `text` is initialized outside of the loop, and you have access to its value. The result of this simple code is shown in Figure 6-6.

FIGURE 6-6

The last value assigned to `text` in the loop is accessible from outside the loop. As you can see, this topic requires a bit of effort to come to grips with. It is not immediately obvious why, in light of the earlier example, `text` doesn't retain the empty string it is assigned before the loop in the code after the loop.

The explanation for this behavior is related to memory allocation for the `text` variable, and indeed any variable. Merely declaring a simple variable type doesn't result in very much happening. It is only when values are assigned to the variables that values are allocated a place in memory to be stored. When this allocation takes place inside a loop, the value is essentially defined as a local value and goes out of scope outside of the loop.

Even though the variable itself isn't localized to the loop, the value it contains is. However, assigning a value outside of the loop ensures that the value is local to the main code, and is still in scope inside the loop. This means that the variable doesn't go out of scope before the main code block is exited, so you have access to its value outside of the loop.

Luckily for you, the C# compiler detects variable scope problems, and responding to the error messages it generates certainly helps you to understand the topic of variable scope.

Finally, be aware of best practices. In general, it is worth declaring and initializing all variables before any code blocks that use them. An exception to this is when you declare looping variables as part of a loop block:

```
for (int i = 0; i < 10; i++)
{
   ...
}
```

Here, `i` is localized to the looping code block, but that's fine because you will rarely require access to this counter from external code.

Parameters and Return Values versus Global Data

Let's take a closer look at exchanging data with functions via global data and via parameters and return values. To recap, consider the following code:

```
class Program
{
   static void ShowDouble(ref int val)
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }

   static void Main(string[] args)
   {
      int val = 5;
      Console.WriteLine("val = {0}", val);
      ShowDouble(ref val);
      Console.WriteLine("val = {0}", val);
   }
}
```

NOTE *This code is slightly different from the code shown earlier in this chapter, when you used the variable name* `myNumber` *in* `Main()`*. This illustrates the fact that local variables can have identical names and yet not interfere with each other. It also means that the two code samples shown here are more similar, enabling you to focus on the specific differences without worrying about variable names.*

Compare it with this code:

```
class Program
{
   static int val;

   static void ShowDouble()
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }
```

```
        static void Main(string[] args)
        {
            val = 5;
            Console.WriteLine("val = {0}", val);
            ShowDouble();
            Console.WriteLine("val = {0}", val);
        }
    }
```

The results of both of these `ShowDouble()` functions are identical.

There are no hard-and-fast rules for using one technique rather than another, and both techniques are perfectly valid, but you may want to consider the following guidelines.

To start with, as mentioned when this topic was first introduced, the `ShowDouble()` version that uses the global value only uses the global variable `val`. To use this version, you must use this global variable. This limits the versatility of the function slightly and means that you must continuously copy the global variable value into other variables if you intend to store the results. In addition, global data might be modified by code elsewhere in your application, which could cause unpredictable results (values might change without you realizing it until it's too late).

However, this loss of versatility can often be a bonus. Sometimes you only want to use a function for one purpose, and using a global data store reduces the possibility that you will make an error in a function call, perhaps passing it the wrong variable.

Of course, it could also be argued that this simplicity actually makes your code more difficult to understand. Explicitly specifying parameters enables you to see at a glance what is changing. If you see a call that reads `FunctionName(val1, out val2)`, you instantly know that `val1` and `val2` are the important variables to consider and that `val2` will be assigned a new value when the function is completed. Conversely, if this function took no parameters, then you would be unable to make any assumptions about what data it manipulated.

Finally, remember that using global data isn't always possible. Later in this book, you will see code written in different files and/or belonging to different namespaces communicating with each other via functions. In these cases, the code is often separated to such a degree that there is no obvious choice for a global storage location.

Feel free to use either technique to exchange data. In general, use parameters rather than global data; but there are certainly cases where global data might be more suitable, and it certainly isn't an error to use that technique.

THE MAIN() FUNCTION

Now that you've covered most of the simple techniques used in the creation and use of functions, it's time to take a closer look at the `Main()` function.

Earlier, you saw that `Main()` is the entry point for a C# application and that execution of this function encompasses the execution of the application. That is, when execution is initiated, the `Main()` function executes, and when the `Main()` function finishes, execution ends.

The `Main()` function can return either `void` or `int`, and can optionally include a `string[] args` parameter, so you can use any of the following versions:

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

The third and fourth versions return an `int` value, which can be used to signify how the application terminates, and often is used as an indication of an error (although this is by no means mandatory). In general, returning a value of 0 reflects normal termination (that is, the application has completed and can terminate safely).

The optional `args` parameter of `Main()` provides you with a way to obtain information from outside the application, specified at runtime. This information takes the form of *command-line parameters*.

You may well have come across command-line parameters already. When you execute an application from the command line, you can often specify information directly, such as a file to load on application execution. For example, consider the Notepad application in Windows. You can run Notepad simply by typing **Notepad** in a command prompt window or in the window that appears when you select the Run option from the Windows Start menu. You can also type something like **Notepad "myfile.txt"** in these locations. The result is that Notepad will either load the file `myfile.txt` when it runs or offer to create this file if it doesn't already exist. Here, "myfile.txt" is a command-line argument. You can write console applications that work similarly by making use of the `args` parameter.

When a console application is executed, any specified command-line parameters are placed in this `args` array. You can then use these parameters in your application. The following Try It Out shows this in action. You can specify any number of command-line arguments, each of which will be output to the console.

TRY IT OUT Command-Line Arguments

1. Create a new console application called Ch06Ex04 and save it in the directory `C:\BegVCSharp\Chapter06`.
2. Add the following code to `Program.cs`:


```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("{0} command line arguments were specified:",
                        args.Length);
      foreach (string arg in args)
         Console.WriteLine(arg);
      Console.ReadKey();
   }
}
```

Code snippet Ch06Ex04\Program.cs

3. Open the property pages for the project (right-click on the Ch06Ex04 project name in the Solution Explorer window and select Properties).

4. Select the Debug page and add any command-line arguments you want to the Command line arguments setting. Figure 6-7 shows an example.

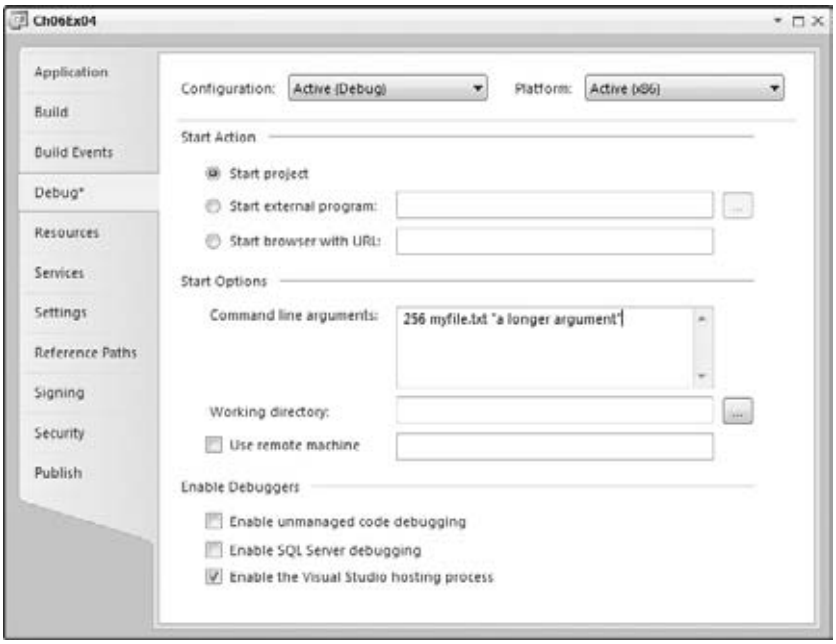

FIGURE 6-7

5. Run the application. Figure 6-8 shows the output.

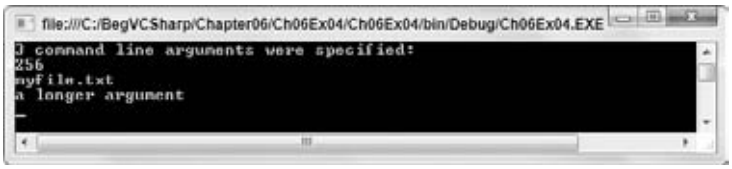

FIGURE 6-8

How It Works

The code used here is very simple:

```
Console.WriteLine("{0} command line arguments were specified:",
                  args.Length);
foreach (string arg in args)
   Console.WriteLine(arg);
```

You're just using the `args` parameter as you would any other string array. You're not doing anything fancy with the arguments; you're just writing whatever is specified to the screen. You supplied the arguments via

the project properties in the IDE. This is a handy way to use the same command-line arguments whenever you run the application from the IDE, rather than type them at a command-line prompt every time. The same result can be obtained by opening a command prompt window in the same directory as the project output (`C:\BegCSharp\Chapter06\Ch06Ex04\Ch06Ex04\bin\Debug`) and typing this:

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

Each argument is separated from the next by spaces. To supply an argument that includes spaces, you can enclose it in double quotation marks, which prevents it from being interpreted as multiple arguments.

STRUCT FUNCTIONS

The last chapter covered struct types for storing multiple data elements in one place. Structs are actually capable of a lot more than this. For example, they can contain functions as well as data. That may seem a little strange at first, but it is, in fact, very useful. As a simple example, consider the following struct:

```
struct CustomerName
{
   public string firstName, lastName;
}
```

If you have variables of type `CustomerName` and you want to output a full name to the console, you are forced to build the name from its component parts. You might use the following syntax for a `CustomerName` variable called `myCustomer`, for example:

```
CustomerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", myCustomer.firstName, myCustomer.lastName);
```

By adding functions to structs, you can simplify this by centralizing the processing of common tasks. For example, you can add a suitable function to the struct type as follows:

```
struct CustomerName
{
   public string firstName, lastName;

   public string Name()
   {
      return firstName + " " + lastName;
   }
}
```

This looks much like any other function you've seen in this chapter, except that you haven't used the `static` modifier. The reasons for this will become clear later in the book; for now, it is enough to know that this keyword isn't required for struct functions. You can use this function as follows:

```
CustomerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(myCustomer.Name());
```

This syntax is much simpler, and much easier to understand, than the previous syntax. The `Name()` function has direct access to the `firstName` and `lastName` struct members. Within the `customerName` struct, they can be thought of as global.

OVERLOADING FUNCTIONS

Earlier in this chapter, you saw how you must match the signature of a function when you call it. This implies that you need to have separate functions to operate on different types of variables. Function overloading provides you with the capability to create multiple functions with the same name, but each working with different parameter types. For example, earlier you used the following code, which contains a function called `MaxValue()`:

```
class Program
{
   static int MaxValue(int[] intArray)
   {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
         if (intArray[i] > maxVal)
            maxVal = intArray[i];
      }
      return maxVal;
   }

   static void Main(string[] args)
   {
      int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
      int maxVal = MaxValue(myArray);
      Console.WriteLine("The maximum value in myArray is {0}", maxVal);
      Console.ReadKey();
   }
}
```

This function can only be used with arrays of `int` values. You could provide different named functions for different parameter types, perhaps renaming the preceding function as `IntArrayMaxValue()` and adding functions such as `DoubleArrayMaxValue()` to work with other types. Alternatively, you could just add the following function to your code:

```
...
static double MaxValue(double[] doubleArray)
{
   double maxVal = doubleArray[0];
   for (int i = 1; i < doubleArray.Length; i++)
   {
      if (doubleArray[i] > maxVal)
         maxVal = doubleArray[i];
   }
   return maxVal;
}
...
```

The difference here is that you are using `double` values. The function name, `MaxValue()`, is the same, but (crucially) its *signature* is different. That's because the signature of a function, as shown earlier, includes both the name of the function and its parameters. It would be an error to define two functions with the same signature, but because these two functions have different signatures, this is fine.

NOTE *The return type of a function isn't part of its signature, so you can't define two functions that differ only in return type; they would have identical signatures.*

After adding the preceding code, you have two versions of `MaxValue()`, which accept `int` and `double` arrays, returning an `int` or `double` maximum, respectively.

The beauty of this type of code is that you don't have to explicitly specify which of these two functions you want to use. You simply provide an array parameter, and the correct function is executed depending on the type of parameter used.

Note another aspect of the IntelliSense feature in VS and VCE: When you have the two functions shown previously in an application and then proceed to type the name of the function, for example, `Main()`, the IDE shows you the available overloads for that function. For example, if you type

```
double result = MaxValue(
```

the IDE gives you information about both versions of `MaxValue()`, which you can scroll between using the up and down arrow keys, as shown in Figure 6-9.

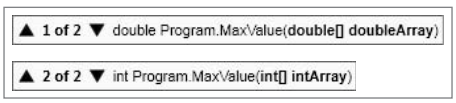

FIGURE 6-9

All aspects of the function signature are included when overloading functions. You might, for example, have two different functions that take parameters by value and by reference, respectively:

```
static void ShowDouble(ref int val)
{
   ...
}
static void ShowDouble(int val)
{
   ...
}
```

Deciding which version to use is based purely on whether the function call contains the `ref` keyword. The following would call the reference version:

```
ShowDouble(ref val);
```

This would call the value version:

```
ShowDouble(val);
```

Alternatively, you could have functions that differ in the number of parameters they require, and so on.

DELEGATES

A *delegate* is a type that enables you to store references to functions. Although this sounds quite involved, the mechanism is surprisingly simple. The most important purpose of delegates will become clear later in the book when you look at events and event handling, but it will be useful to briefly consider them here. Delegates are declared much like functions, but with no function body and using the `delegate` keyword. The delegate declaration specifies a return type and parameter list.

After defining a delegate, you can declare a variable with the type of that delegate. You can then initialize the variable as a reference to any function that has the same return type and parameter list as that delegate. Once you have done this, you can call that function by using the delegate variable as if it were a function.

When you have a variable that refers to a function, you can also perform other operations that would be otherwise impossible. For example, you can pass a delegate variable to a function as a parameter, and then that function can use the delegate to call whatever function it refers to, without knowing what function will be called until runtime. The following Try It Out demonstrates using a delegate to access one of two functions.

TRY IT OUT Using a Delegate to Call a Function

1. Create a new console application called Ch06Ex05 and save it in the directory `C:\BegVCSharp\Chapter06`.
2. Add the following code to `Program.cs`:


```
class Program
{
   delegate double ProcessDelegate(double param1, double param2);

   static double Multiply(double param1, double param2)
   {
      return param1 * param2;
   }

   static double Divide(double param1, double param2)
   {
      return param1 / param2;
   }

   static void Main(string[] args)
   {
      ProcessDelegate process;
      Console.WriteLine("Enter 2 numbers separated with a comma:");
      string input = Console.ReadLine();
      int commaPos = input.IndexOf(',');
      double param1 = Convert.ToDouble(input.Substring(0, commaPos));
      double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                       input.Length - commaPos - 1));
      Console.WriteLine("Enter M to multiply or D to divide:");
```

```
            input = Console.ReadLine();
            if (input == "M")
               process = new ProcessDelegate(Multiply);
            else
               process = new ProcessDelegate(Divide);
            Console.WriteLine("Result: {0}", process(param1, param2));
            Console.ReadKey();
         }
      }
```

Code snippet Ch06Ex05\Program.cs

3. Execute the code. Figure 6-10 shows the result.

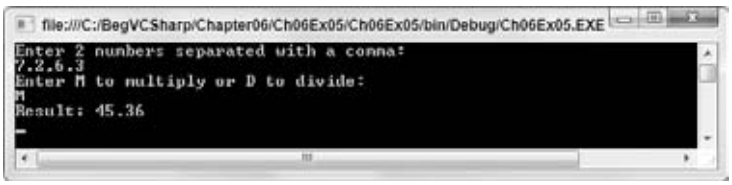

FIGURE 6-10

How It Works

This code defines a delegate (`ProcessDelegate`) whose return type and parameters match those of the two functions (`Multiply()` and `Divide()`). The delegate definition is as follows:

```
delegate double ProcessDelegate(double param1, double param2);
```

The `delegate` keyword specifies that the definition is for a delegate, rather than a function (the definition appears in the same place that a function definition might). Next, the definition specifies a `double` return value and two `double` parameters. The actual names used are arbitrary; you can call the delegate type and parameter names whatever you like. Here, we've used a delegate name of `ProcessDelegate` and double parameters called `param1` and `param2`.

The code in `Main()` starts by declaring a variable using the new delegate type:

```
static void Main(string[] args)
{
   ProcessDelegate process;
```

Next, you have some fairly standard C# code that requests two numbers separated by a comma, and then places these numbers in two `double` variables:

```
   Console.WriteLine("Enter 2 numbers separated with a comma:");
   string input = Console.ReadLine();
   int commaPos = input.IndexOf(',');
   double param1 = Convert.ToDouble(input.Substring(0, commaPos));
   double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                    input.Length -- commaPos -- 1));
```

NOTE *For demonstration purposes, no user input validation is included here. If this were "real" code, you'd spend much more time ensuring that you had valid values in the local* `param1` *and* `param2` *variables.*

Next, you ask the user to multiply or divide these numbers:

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

Based on the user's choice, you initialize the `process` delegate variable:

```
if (input == "M")
   process = new ProcessDelegate(Multiply);
else
   process = new ProcessDelegate(Divide);
```

To assign a function reference to a delegate variable, you use slightly odd-looking syntax. Much like assigning array values, you must use the `new` keyword to create a new delegate. After this keyword, you specify the delegate type and supply a parameter referring to the function you want to use — namely, the `Multiply()` or `Divide()` function. This parameter doesn't match the parameters of the delegate type or the target function; it is a syntax unique to delegate assignment. The parameter is simply the name of the function to use, without any parentheses.

In fact, you can use slightly simpler syntax here, if you want:

```
if (input == "M")
   process = Multiply;
else
   process = Divide;
```

The compiler recognizes that the delegate type of the process variable matches the signature of the two functions, and automatically initializes a delegate for you. Which syntax you use is up to you, although some people prefer to use the longhand version, as it is easier to see at a glance what is happening.

Finally, call the chosen function using the delegate. The same syntax works, regardless of which function the delegate refers to:

```
   Console.WriteLine("Result: {0}", process(param1, param2));
   Console.ReadKey();
}
```

Here, you treat the delegate variable as if it were a function name. Unlike a function, though, you can also perform additional operations on this variable, such as passing it to a function via a parameter, as shown in this simple example:

```
static void ExecuteFunction(ProcessDelegate process)
{
   process(2.2, 3.3);
}
```

This means that you can control the behavior of functions by passing them function delegates, much like choosing a "snap-in" to use. For example, you might have a function that sorts a string array alphabetically. You can use several techniques to sort lists, with varying performance depending on the

characteristics of the list being sorted. By using delegates, you can specify the function to use by passing a sorting algorithm function delegate to a sorting function.

There are many such uses for delegates, but, as mentioned earlier, their most prolific use is in event handling, covered in Chapter 13.

SUMMARY

This chapter provided a fairly complete overview of the use of functions in C# code. Many of the additional features that functions offer (delegates in particular) are more abstract, and you need to understand them in regard to object-oriented programming, the subject of Chapter 8.

Knowing how to use functions is central to all of the programming you are likely to do. Later chapters, particularly when you get to OOP (from Chapter 8 onward), explain a more formal structure for functions and how they apply to classes. You will likely find that the capability to abstract code into reusable blocks is the most useful aspect of C# programming.

EXERCISES

1. The following two functions have errors. What are they?

```
static bool Write()
{
   Console.WriteLine("Text output from function.");
}

static void MyFunction(string label, params int[] args, bool showLabel)
{
   if (showLabel)
      Console.WriteLine(label);
   foreach (int i in args)
      Console.WriteLine("{0}", i);
}
```

2. Write an application that uses two command-line arguments to place values into a string and an integer variable, respectively. Then display those values.

3. Create a delegate and use it to impersonate the `Console.ReadLine()` function when asking for user input.

4. Modify the following struct to include a function that returns the total price of an order:

```
struct order
{
   public string itemName;
   public int    unitCount;
   public double unitCost;
}
```

5. Add another function to the order struct that returns a formatted string as follows (as a single line of text, where italic entries enclosed in angle brackets are replaced by appropriate values):

```
Order Information: <unit count> <item name> items at $<unit cost> each,
total cost $<total cost>
```

Answers to Exercises can be found in Appendix A.

► WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Defining functions	Functions are defined with a name, zero or more parameters, and a return type. The name and parameters of a function collectively define the signature of the function. It is possible to define multiple functions whose signatures are different even though their names are the same — this is called function overloading. Functions can also be defined within `struct` types.
Return values and parameters	The return type of a function can be any type, or `void` if the function does not return a value. Parameters can also be of any type, and consist of a comma-separated list of type and name pairs. When calling a function, any parameters specified must match those in the definition both in type and in order. A variable number of parameters of a specified type can be specified through a parameter array. Parameters can be specified as `ref` or `out` parameters in order to return values to the caller.
Variable scope	Variables are scoped according to the block of code where they are defined. Blocks of code include methods as well as other structures, such as the body of a loop. It is possible to define multiple, separate variables with the same name at different scope levels.
Command-line parameters	The `Main()` function in a console application can receive command-line parameters that are passed to the application when it is executed. These parameters are separated by spaces, but longer parameters can be passed in quotes.
Delegates	As well as calling functions directly, it is possible to call them through delegates. Delegates are variables that are defined with a return type and parameter list. A given delegate type can match any method whose return type and parameters match the delegate definition.

7

Debugging and Error Handling

WHAT YOU WILL LEARN IN THIS CHAPTER

- Debugging methods available in the IDE
- Error-handling techniques available in C#

So far this book has covered all the basics of simple programming in C#. Before you move on to object-oriented programming in the next part, you need to look at debugging and error handling in C# code.

Errors in code are something that will always be with you. No matter how good a programmer is, problems will always slip through, and part of being a good programmer is realizing this and being prepared to deal with it. Of course, some problems are minor and don't affect the execution of an application, such as a spelling mistake on a button, but glaring errors are also possible, including those that cause applications to fail completely (usually known as *fatal errors*). Fatal errors include simple errors in code that prevent compilation (syntax errors), or more serious problems that occur only at runtime. Some errors are subtle. Perhaps your application fails to add a record to a database because a requested field is missing, or adds a record with the wrong data in other restricted circumstances. Errors such as these, where application logic is in some way flawed, are known as *semantic errors*, or *logic errors*.

Often, you won't know about these subtle errors until a user of your application complains that something isn't working properly. This leaves you with the task of tracing through your code to find out what's happening and fixing it so that it does what it was intended to do. In these situations, the debugging capabilities of VS and VCE are a fantastic help. The first part of this chapter looks at some of the techniques available and applies them to some common problems.

Then, you'll learn the error-handling techniques available in C#. These enable you to take precautions in cases where errors are likely, and to write code that is resilient enough to cope with errors that might otherwise be fatal. The techniques are part of the C# language, rather than a debugging feature, but the IDE provides some tools to help you here, too.

DEBUGGING IN VS AND VCE

Earlier, you learned that you can execute applications in two ways: with debugging enabled or without debugging enabled. By default, when you execute an application from VS or VCE, it executes with debugging enabled. This happens, for example, when you press F5 or click the green Play arrow in the toolbar. To execute an application without debugging enabled, choose Debug ➪ Start Without Debugging, or press Ctrl+F5.

Both VS and VCE allow you to build applications in two configurations: Debug (the default) and Release. (In fact, you can define additional configurations, but that's an advanced technique not covered here.) You can switch between these configurations using the Solution Configurations drop-down in the Standard toolbar.

NOTE *In VCE the Solution Configurations drop-down list is inactive by default. To work through this chapter, enable it by selecting Tools ➪ Options. In the Options dialog, ensure that Show All Settings is selected, choose the General subcategory of the Projects and Solutions category, and enable the Show Advanced Build Configurations option.*

When you build an application in debug configuration and execute it in debug mode, more is going on than the execution of your code. Debug builds maintain *symbolic information* about your application, so that the IDE knows exactly what is happening as each line of code is executed. Symbolic information means keeping track of, for example, the names of variables used in uncompiled code, so they can be matched to the values in the compiled machine code application, which won't contain such human-readable information. This information is contained in `.pdb` files, which you may have seen in your computer's `Debug` directories. This enables you to perform many useful operations:

- Outputting debugging information to the IDE
- Looking at (and editing) the values of variables in scope during application execution
- Pausing and restarting program execution
- Automatically halting execution at certain points in the code
- Stepping through program execution one line at a time
- Monitoring changes in variable content during application execution
- Modifying variable content at runtime
- Performing test calls of functions

In the release configuration, application code is optimized, and you cannot perform these operations. However, release builds also run faster; and when you have finished developing an application, you will typically supply users with release builds because they won't require the symbolic information that debug builds include.

This section describes debugging techniques you can use to identify and fix areas of code that don't work as expected, a process known as *debugging*. The techniques are grouped into two sections according to how they are used. In general, debugging is performed either by interrupting program

execution or by making notes for later analysis. In VS and VCE terms, an application is either running or in break mode — that is, normal execution is halted. You'll look at the nonbreak mode (runtime or normal) techniques first.

Debugging in Nonbreak (Normal) Mode

One of the commands you've been using throughout this book is the `Console.WriteLine()` function, which outputs text to the console. As you are developing applications, this function comes in handy for getting extra feedback about operations:

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc("Do something.");
Console.WriteLine("MyFunc() Function execution completed.");
```

This code snippet shows how you can get extra information concerning a function called `MyFunc()`. This is all very well, but it can make your console output a bit cluttered; and when you develop other types of applications, such as Windows Forms applications, you won't have a console to output information to. As an alternative, you can output text to a separate location — the Output window in the IDE.

Chapter 2, which describes the Error List window, mentions that other windows can also be displayed in the same place. One of these, the Output window, can be very useful for debugging. To display this window, select View ➪ Output. This window provides information related to compilation and execution of code, including errors encountered during compilation. You can also use this window, shown in Figure 7-1, to display custom diagnostic information by writing to it directly.

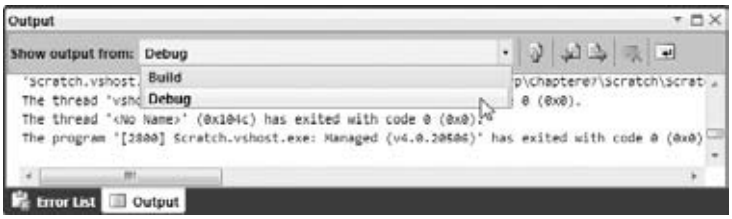

FIGURE 7-1

NOTE *The Output window contains a drop-down menu from which different modes can be selected, including Build and Debug. These modes display compilation and runtime information, respectively. When you read "writing to the Output window" in this section, it actually means "writing to the debug mode view of the Output window."*

Alternatively, you might want to create a logging file, which has information appended to it when your application is executed. The techniques for doing this are much the same as those for writing text to the Output window, although the process requires an understanding of how to access the file system from C# applications. For now, leave that functionality on the back burner because there is plenty you can do without getting bogged down by file-access techniques.

Outputting Debugging Information

Writing text to the Output window at runtime is easy. You simply replace calls to `Console.WriteLine()` with the required call to write text where you want it. There are two commands you can use to do this:

- `Debug.WriteLine()`
- `Trace.WriteLine()`

These commands function in almost exactly the same way — with one key difference: The first command works in debug builds only; the latter works for release builds as well. In fact, the `Debug.WriteLine()` command won't even be compiled into a release build; it just disappears, which certainly has its advantages (the compiled code will be smaller, for one thing). You can, in effect, create two versions of your application from a single source file. The debug version displays all kinds of extra diagnostic information, whereas the release version won't have this overhead, and won't display messages to users that might otherwise be annoying!

These functions don't work exactly like `Console.WriteLine()`. They work with only a single string parameter for the message to output, rather than letting you insert variable values using `{X}`syntax. This means you must use an alternative technique to embed variable values in strings — for example, the `+` concatenation. You can also (optionally) supply a second string parameter, which displays a category for the output text. This enables you to see at a glance what output messages are displayed in the Output window, which is useful when similar messages are output from different places in the application.

The general output of these functions is as follows:

```
<category>: <message>
```

For example, the following statement, which has `"MyFunc"` as the optional category parameter,

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

would result in the following:

```
MyFunc: Added 1 to i
```

The next Try It Out demonstrates outputting debugging information in this way.

TRY IT OUT Writing Text to the Output Window

1. Create a new console application called Ch07Ex01 and save it in the directory `C:\BegVCSharp\Chapter07`.
2. Modify the code as follows:


```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
```

```
namespace Ch07Ex01
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
         int[] maxValIndices;
         int maxVal = Maxima(testArray, out maxValIndices);
         Console.WriteLine("Maximum value {0} found at element indices:",
                           maxVal);
         foreach (int index in maxValIndices)
         {
            Console.WriteLine(index);
         }
         Console.ReadKey();
      }
      static int Maxima(int[] integers, out int[] indices)
      {
         Debug.WriteLine("Maximum value search started.");
         indices = new int[1];
         int maxVal = integers[0];
         indices[0] = 0;
         int count = 1;
         Debug.WriteLine(string.Format(
            "Maximum value initialized to {0}, at element index 0.", maxVal));
         for (int i = 1; i < integers.Length; i++)
         {
            Debug.WriteLine(string.Format(
               "Now looking at element at index {0}.", i));
            if (integers[i] > maxVal)
            {
               maxVal = integers[i];
               count = 1;
               indices = new int[1];
               indices[0] = i;
               Debug.WriteLine(string.Format(
                  "New maximum found. New value is {0}, at element index {1}.",
                  maxVal, i));
            }
            else
            {
               if (integers[i] == maxVal)
               {
                  count++;
                  int[] oldIndices = indices;
                  indices = new int[count];
                  oldIndices.CopyTo(indices, 0);
                  indices[count - 1] = i;
                  Debug.WriteLine(string.Format(
                     "Duplicate maximum found at element index {0}.", i));
               }
            }
         }
```

```
            Trace.WriteLine(string.Format(
                        "Maximum value {0} found, with {1} occurrences.", maxVal, count));
                    Debug.WriteLine("Maximum value search completed.");
                    return maxVal;
                }
            }
        }
```

Code snippet Ch07Ex01\Program.cs

3. Execute the code in debug mode. The result is shown in Figure 7-2.

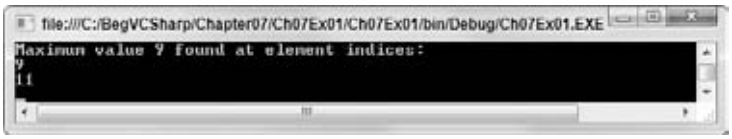

FIGURE 7-2

4. Terminate the application and check the contents of the Output window (in debug mode). A truncated version of the output is shown here:

```
...

Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.
Now looking at element at index 4.
Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The thread 'vshost.RunParkingWindow' (0x110c) has exited with code 0 (0x0).
The thread '<No Name>' (0x688) has exited with code 0 (0x0).
The program '[4568] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with
code 0 (0x0).
```

5. Change to release mode using the drop-down menu on the Standard toolbar, as shown in Figure 7-3.

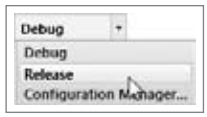

FIGURE 7-3

6. Run the program again, this time in release mode, and recheck the Output window when execution terminates. The output (again truncated) is as follows:

```
...
Maximum value 9 found, with 2 occurrences.
The thread 'vshost.RunParkingWindow' (0xa78) has exited with code 0 (0x0).
The thread '<No Name>' (0x130c) has exited with code 0 (0x0).
The program '[4348] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with
code 0 (0x0).
```

How It Works

This application is an expanded version of one shown in Chapter 6, using a function to calculate the maximum value in an integer array. This version also returns an array of the indices where maximum values are found in an array, so that the calling code can manipulate these elements.

First, an additional `using` directive appears at the beginning of the code:

```
using System.Diagnostics;
```

This simplifies access to the functions discussed earlier because they are contained in the `System.Diagnostics` namespace. Without this `using` directive, code such as

```
Debug.WriteLine("Bananas");
```

would need further qualification, and would have to be rewritten as

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

The `using` directive keeps your code simple and reduces verbosity.

The code in `Main()` simply initializes a test array of integers called `testArray`; it also declares another integer array called `maxValIndices` to store the index output of `Maxima()` (the function that performs the calculation), and then calls this function. Once the function returns, the code simply outputs the results.

`Maxima()` is slightly more complicated, but it doesn't use much code that you haven't already seen. The search through the array is performed in a similar way to the `MaxVal()` function in Chapter 6, but a record is kept of the indices of maximum values.

Especially note (other than the lines that output debugging information) the function used to keep track of the indices. Rather than return an array that would be large enough to store every index in the source array (needing the same dimensions as the source array), `Maxima()` returns an array just large enough to hold the indices found. It does this by continually recreating arrays of different sizes as the search progresses. This is necessary because arrays can't be resized once they are created.

The search is initialized by assuming that the first element in the source array (called `integers` locally) is the maximum value and that there is only one maximum value in the array. Values can therefore be set for `maxVal` (the return value of the function and the maximum value found) and `indices`, the `out` parameter array that stores the indices of the maximum values found. `maxVal` is assigned the value of the first element

in `integers`, and `indices` is assigned a single value, simply `0`, which is the index of the array's first element. You also store the number of maximum values found in a variable called `count`, which enables you to keep track of the `indices` array.

The main body of the function is a loop that cycles through the values in the `integers` array, omitting the first one because it has already been processed. Each value is compared to the current value of `maxVal` and ignored if `maxVal` is greater. If the currently inspected array value is greater than `maxVal`, then `maxVal` and `indices` are changed to reflect this. If the value is equal to `maxVal`, then `count` is incremented and a new array is substituted for `indices`. This new array is one element bigger than the old `indices` array, containing the new index found.

The code for this last piece of functionality is as follows:

```
if (integers[i] == maxVal)
{
   count++;
   int[] oldIndices = indices;
   indices = new int[count];
   oldIndices.CopyTo(indices, 0);
   indices[count - 1] = i;
   Debug.WriteLine(string.Format(
      "Duplicate maximum found at element index {0}.", i));
}
```

This works by backing up the old `indices` array into `oldIndices`, an integer array local to this `if` code block. Note that the values in `oldIndices` are copied into the new `indices` array using the `<array>.CopyTo()` function. This function simply takes a target array and an index to use for the first element to copy to and pastes all values into the target array.

Throughout the code, various pieces of text are output using the `Debug.WriteLine()` and `Trace.WriteLine()` functions. These functions use the `string.Format()` function to embed variable values in strings in the same way as `Console.WriteLine()`. This is slightly more efficient than using the `+` concatenation operator.

When you run the application in debug mode, you see a complete record of the steps taken in the loop that give you the result. In release mode, you see just the result of the calculation, because no calls to `Debug.WriteLine()` are made in release builds.

In addition to these `WriteLine()` functions, there are a few more you should be aware of. To start with, there are equivalents to `Console.Write()`:

- `Debug.Write()`
- `Trace.Write()`

Both functions use the same syntax as the `WriteLine()` functions (one or two parameters, with a message and an optional category), but differ in that they don't add end-of-line characters.

There are also the following commands:

- `Debug.WriteLineIf()`
- `Trace.WriteLineIf()`

- `Debug.WriteIf()`
- `Trace.WriteIf()`

Each of these has the same parameters as the non-`If` counterparts, with the addition of an extra, mandatory parameter that precedes them in the parameter list. This parameter takes a Boolean value (or an expression that evaluates to a Boolean value) and results in the function only writing text if this value evaluates to `true`. You can use these functions to conditionally output text to the Output window.

For example, you might require debugging information to be output in only certain situations, so you can have a great many `Debug.WriteLineIf()` statements in your code that all depend on a certain condition being met. If this condition doesn't occur, then they aren't displayed, which prevents the Output window from being cluttered with superfluous information.

Tracepoints

An alternative to writing information to the Output window is to use *tracepoints*. These are a feature of VS, rather than C#, but they serve the same function as using `Debug.WriteLine()`. Essentially, they enable you to output debugging information without modifying your code.

NOTE *Tracepoints are a feature only available in VS, not in VCE. If you are using VCE, you may choose to skip this section.*

To demonstrate tracepoints, you can use them to replace the debugging commands in the previous example. (See the `Ch07Ex01TracePoints` file in the downloadable code for this chapter.) The process for adding a tracepoint is as follows:

1. Position the cursor at the line where you want the tracepoint to be inserted. The tracepoint will be processed *before* this line of code is executed.
2. Right-click the line of code and select Breakpoint ➪ Insert Tracepoint.
3. Type the string to be output in the Print a Message text box in the When Breakpoint Is Hit dialog that appears. If you want to output variable values, enclose the variable name in curly braces.
4. Click OK. A red diamond appears to the left of the line of code containing a tracepoint, and the line of code itself is shown with red highlighting.

As implied by the title of the dialog for adding tracepoints, and the menu selections required for them, tracepoints are a form of breakpoint (and can cause application execution to pause, just like a breakpoint, if desired). You look at breakpoints, which typically serve a more advanced debugging purpose, a little later in the chapter.

Figure 7-4 shows the tracepoint required for line 31 of `Ch07Ex01TracePoints`, where line numbering applies to the code after the existing `Debug.WriteLine()` statements have been removed.

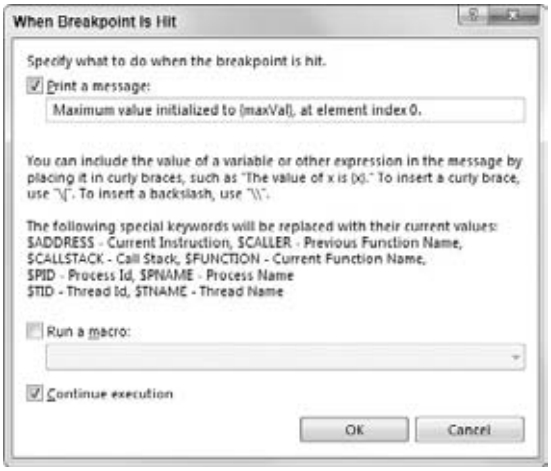

FIGURE 7-4

NOTE *As shown in the text in Figure 7-4, tracepoints enable you to insert other useful information concerning the location and context of the tracepoint. Experiment with these values, particularly* `$FUNCTION` *and* `$CALLER`*, to see what additional information you can glean. You can also see that it is possible for the tracepoint to execute a macro, an advanced feature that isn't covered here.*

There is another window (only available in VS) that you can use to quickly see the tracepoints in an application. To display this window, select Debug ➪ Windows ➪ Breakpoints from the VS menu. This is a general window for displaying breakpoints (tracepoints, as noted earlier, are a form of breakpoint). You can customize the display to show more tracepoint-specific information by adding the When Hit column from the Columns drop-down in this window. Figure 7-5 shows the display with this column configured and all the tracepoints added to `Ch07Ex01TracePoints`.

Executing this application in debug mode has the same result as before. You can remove or temporarily disable tracepoints by right-clicking on them in the code window or via the Breakpoints window. In the Breakpoints window, the check box to the left of the tracepoint indicates whether the tracepoint is enabled; disabled tracepoints are unchecked and displayed in the code window as diamond outlines, rather than solid diamonds.

Diagnostics Output Versus Tracepoints

Now that you have seen two methods of outputting essentially the same information, consider the pros and cons of each. First, tracepoints have no equivalent to the `Trace` commands; that is, there is no way to output information in a release build using tracepoints. This is because tracepoints are not included in your application. Tracepoints are handled by Visual Studio and, as such, do not exist in the compiled version of your application. You will see tracepoints doing something only when your application is running in the VS debugger.

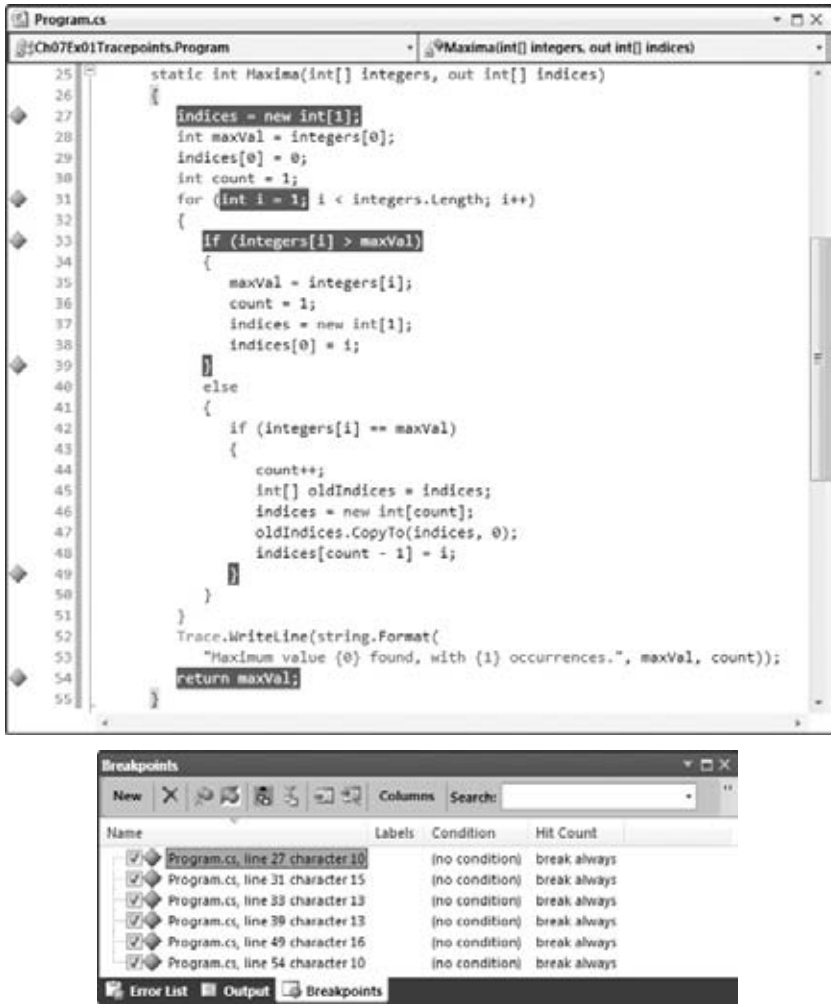

FIGURE 7-5

The chief disadvantage of tracepoints is also their major advantage, which is that they are stored in VS. This makes them quick and easy to add to your applications as and when you need them, but also all too easy to delete. Deleting a tracepoint is as simple as clicking on the red diamond indicating its position, which can be annoying if you are outputting a complicated string of information.

One bonus of tracepoints, though, is the additional information that can be easily added, such as `$FUNCTION`, as noted in the previous section. While this information is available to code written using `Debug` and `Trace` commands, it is trickier to obtain. In summary, use these two methods of outputting debug information as follows:

- **Diagnostics output:** Use when debug output is something you always want to output from an application, particularly where the string you want to output is complex, involving several

variables or a lot of information. In addition, `Trace` commands are often the only option should you want output during execution of an application built in release mode.

- **Tracepoints:** Use these when debugging an application to quickly output important information that may help you resolve semantic errors.

There is also the obvious difference that tracepoints are only available in VS, whereas diagnostics output is available in both VS and VCE.

Debugging in Break Mode

The rest of the debugging techniques described in this chapter work in break mode. This mode can be entered in several ways, all of which result in the program pausing in some way.

Entering Break Mode

The simplest way to enter break mode is to click the Pause button in the IDE while an application is running. This Pause button is found on the Debug toolbar, which you should add to the toolbars that appear by default in VS. To do this, right-click in the toolbar area and select Debug. Figure 7-6 shows the Debug toolbar that appears.

FIGURE 7-6

The first four buttons on the toolbar allow manual control of breaking. In Figure 7-6, three of these are grayed out because they won't work with a program that isn't currently executing. The one that is enabled, Start, is identical to the button that exists on the standard toolbar. The following sections describe the rest of the buttons as needed.

When an application is running, the toolbar changes to look like Figure 7-7.

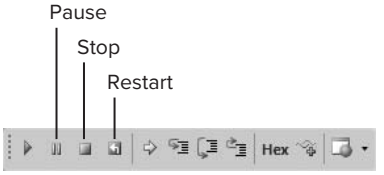

FIGURE 7-7

The three buttons next to Start that were grayed out now enable you to do the following:

- Pause the application and enter break mode.
- Stop the application completely (this doesn't enter break mode, it just quits).
- Restart the application.

Pausing the application is perhaps the simplest way to enter break mode, but it doesn't give you fine-grained control over exactly where to stop. You are likely to stop in a natural pause in the application, perhaps where you request user input. You might also be able to enter break mode during a lengthy operation, or a long loop, but the exact stop point is likely to be fairly random. In general, it is far better to use breakpoints.

Breakpoints

A *breakpoint* is a marker in your source code that triggers automatic entry into break mode. Breakpoints are available in both VS and VCE, but they are more flexible in VS. Breakpoints may be configured to do the following:

- Enter break mode immediately when the breakpoint is reached.

- (VS only) Enter break mode when the breakpoint is reached if a Boolean expression evaluates to `true`.
- (VS only) Enter break mode once the breakpoint is reached a set number of times.
- (VS only) Enter break mode once the breakpoint is reached and a variable value has changed since the last time the breakpoint was reached.
- (VS only) Output text to the Output window or execute a macro (see the section "Tracepoints" earlier in the chapter).

These features are available only in debug builds. If you compile a release build, all breakpoints are ignored.

There are several ways to add breakpoints. To add simple breakpoints that break when a line is reached, just left-click on the far left of the line of code, right-click on the line, and select Breakpoint ➪ Insert Breakpoint; select Debug ➪ Toggle Breakpoint from the menu; or press F9.

A breakpoint appears as a red circle next to the line of code, which is highlighted, as shown in Figure 7-8.

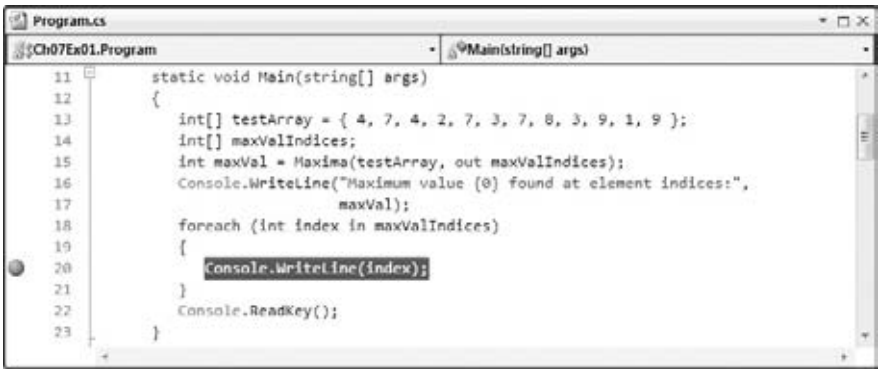

FIGURE 7-8

The remainder of this section applies only to VS, not VCE. If you are using VCE, then feel free to skip ahead to the section "Other Ways to Enter Break Mode."

In VS, you can also see information about a file's breakpoints using the Breakpoints window (you saw how to enable this window earlier). You can use the Breakpoints window to disable breakpoints (by removing the tick to the left of a description; a disabled breakpoint shows up as an unfilled red circle), to delete breakpoints, and to edit the properties of breakpoints.

The columns shown in this window, Condition and Hit Count, are only two of the available ones, but they are the most useful. You can edit these by right-clicking a breakpoint (in code or in this window) and selecting Condition or Hit Count.

Selecting Condition opens a dialog in which you can type any Boolean expression, which may involve any variables in scope at the breakpoint. For example, you could configure a breakpoint that triggers

when it is reached and the value of `maxVal` is greater than 4 by entering the expression `"maxVal > 4"` and selecting the `"Is true"` option. You can also check whether the value of this expression has changed and only trigger the breakpoint then (you might trigger it if `maxVal` changed from 2 to 6 between breakpoint encounters, for example).

Selecting Hit Count opens a dialog in which you can specify how many times a breakpoint needs to be hit before it is triggered. A drop-down list offers the following options:

- Break always
- Break when the hit count is equal to
- Break when the hit count is a multiple of
- Break when the hit count is greater than or equal to

The option chosen, combined with the value entered in the text box next to the options, determines the behavior of the breakpoint. The hit count is useful in long loops, when you might want to break after, say, the first 5,000 cycles. It would be a pain to break and restart 5,000 times if you couldn't do this!

NOTE *A breakpoint with additional properties set (such as a condition or hit count) is displayed slightly differently. Instead of a simple red circle, a configured breakpoint consists of a red circle containing a white + (plus) symbol. This can be useful because it enables you to see at a glance which breakpoints will always cause break mode to be entered and which will only do so in certain circumstances.*

Other Ways to Enter Break Mode

There are two more ways to get into break mode. One is to enter it when an unhandled exception is thrown. This subject is covered later in this chapter, when you look at error handling. The other way is to break when an *assertion* is generated.

Assertions are instructions that can interrupt application execution with a user-defined message. They are often used during application development to test whether things are going smoothly. For example, at some point in your application you might require a given variable to have a value less than 10. You can use an assertion to confirm that this is true, interrupting the program if it isn't. When the assertion occurs, you have the option to Abort, which terminates the application; Retry, which causes break mode to be entered; or Ignore, which causes the application to continue as normal.

As with the debug output functions shown earlier, there are two versions of the assertion function:

- `Debug.Assert()`
- `Trace.Assert()`

Again, the debug version is only compiled into debug builds.

These functions take three parameters. The first is a Boolean value, whereby a value of `false` causes the assertion to trigger. The second and third are string parameters to write information both to a pop-up dialog and the Output window. The preceding example would need a function call such as the following:

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.",
             "Assertion occurred in Main().");
```

Assertions are often useful in the early stages of user adoption of an application. You can distribute release builds of your application containing `Trace.Assert()` functions to keep tabs on things. Should an assertion be triggered, the user will be informed, and this information can be passed on to you. You can then determine what has gone wrong even if you don't know how it went wrong.

You might, for example, provide a brief description of the error in the first string, with instructions as to what to do next as the second string:

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
             "Please contact vendor with the error code KCW001.");
```

Should this assertion occur, the user will see the dialog shown in Figure 7-9.

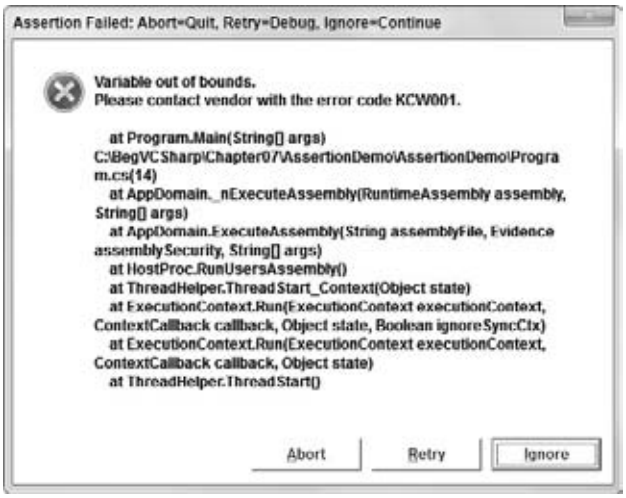

FIGURE 7-9

Admittedly, this isn't the most user-friendly dialog in the world, as it contains a lot of information that could confuse users, but if they send you a screenshot of the error, you could quickly track down the problem.

Now it's time to look at what you can actually do after application execution is halted and you are in break mode. In general, you enter break mode to find an error in your code (or to reassure yourself that things are working properly). Once you are in break mode, you can use various techniques, all of which enable you to analyze your code and the exact state of the application at the point in its execution where it is paused.

Monitoring Variable Content

Monitoring variable content is just one example of how VS and VCE help you a great deal by simplifying things. The easiest way to check the value of a variable is to hover the mouse over its name in the source code while in break mode. A yellow tooltip showing information about the variable appears, including the variable's current value.

You can also highlight entire expressions to get information about their results in the same way. For more complex values, such as arrays, you can even expand values in the tooltip to see individual element entries.

You may have noticed that when you run an application, the layout of the various windows in the IDE changes. By default, the following changes are likely to occur at runtime (this behavior may vary slightly depending on your installation):

- The Properties window disappears, along with some other windows, probably including the Solution Explorer window.
- The Error List window is replaced with two new windows across the bottom of the IDE window.
- Several new tabs appear in the new windows.

The new screen layout is shown in Figure 7-10. This may not match your display exactly, and some of the tabs and windows may not look exactly the same, but the functionality of these windows as described later will be the same, and this display is completely customizable via the View and Debug ➪ Windows menus (during break mode), as well as by dragging windows around the screen to reposition them.

The new window that appears in the bottom-left corner is particularly useful for debugging. It enables you to keep tabs on the values of variables in your application when in break mode. The tabs displayed here vary between VS and VCE:

- **Autos (VS only):** Variables in use in the current and previous statements (Ctrl+D, A)
- **Locals:** All variables in scope (Ctrl+D, L)
- **Watch N:** Customizable variable and expression display (where N is 1 to 4, found on Debug Windows Watch)

All these tabs work in more or less the same way, with various additional features depending on their specific function. In general, each tab contains a list of variables, with information on each variable's name, value, and type. More complex variables, such as arrays, may be further examined using the + and – tree expansion/contraction symbols to the left of their names, enabling a tree view of their content. For example, Figure 7-11 shows the Locals tab obtained by placing a breakpoint in the example code. It shows the expanded view for one of the array variables, `maxValIndices`.

You can also edit the content of variables from this view. This effectively bypasses any other variable assignment that might have happened in earlier code. To do this, simply type a new value into the Value column for the variable you want to edit. You might do this to try out some scenarios that would otherwise require code changes, for example.

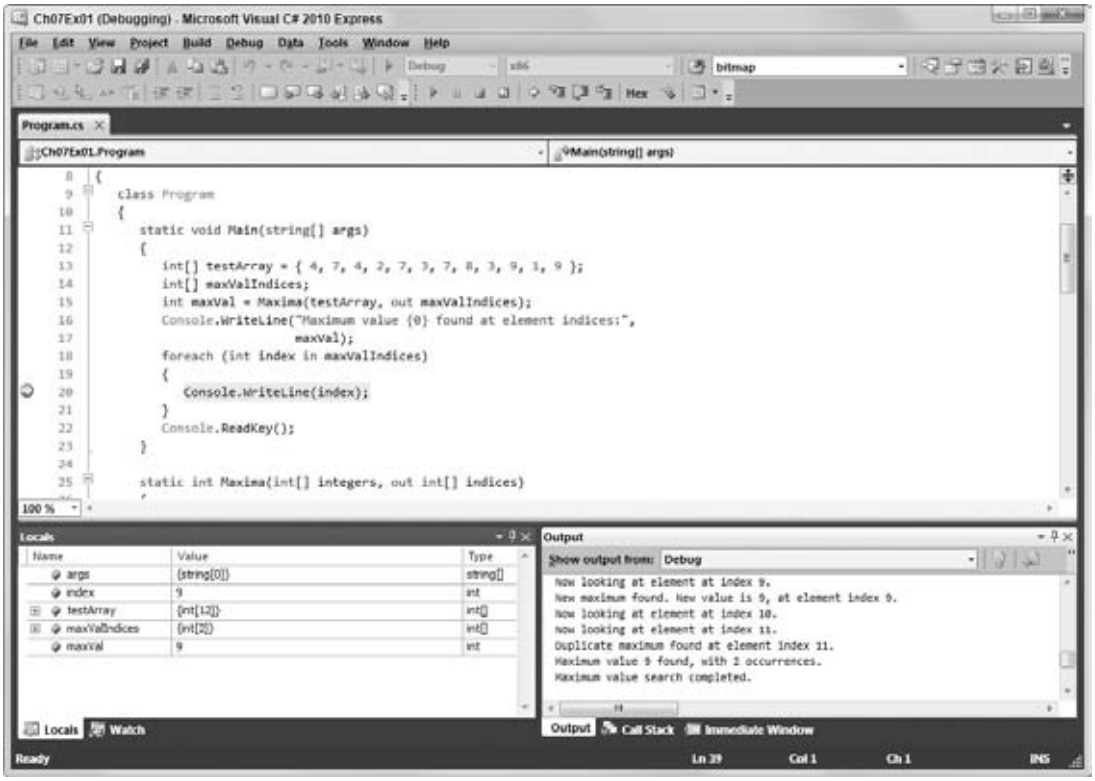

FIGURE 7-10

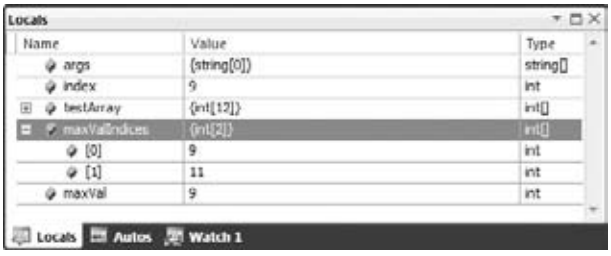

FIGURE 7-11

The Watch window (or Watch windows in VS, which can display up to four) enables you to monitor specific variables, or expressions involving specific variables. To use this window, type the name of a variable or expression into the Name column and view the results. Note that not all variables in an application are in scope all the time, and are labeled as such in a Watch window. For example, Figure 7-12 shows a Watch window with a few sample variables and expressions in it, obtained when a breakpoint just before the end of the `Maxima()` function is reached.

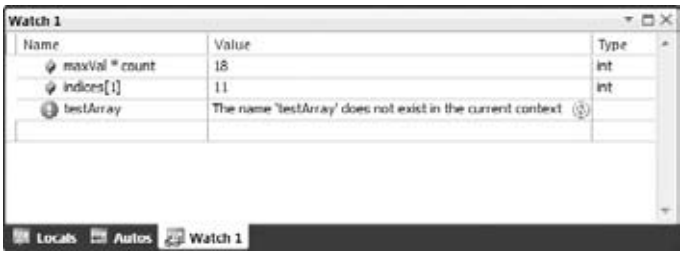

FIGURE 7-12

The `testArray` array is local to `Main()`, so you don't see a value here. Instead, you get a message informing you that the variable isn't in scope.

NOTE *You can also add variables to a Watch window by dragging them from the source code into the window.*

One nice feature about the various displays of variables accessible in this window is that they show you variables that have changed between breakpoints. Any new value is shown in red, rather than black, making it easy to see whether a value has changed.

As mentioned earlier, to add more Watch windows in VS, in break mode you can use the Debug ➪ Windows ➪ Watch ➪ Watch N menu options to toggle the four possible windows on or off. Each window may contain an individual set of watches on variables and expressions, so you can group related variables together for easy access.

As well as these windows, VS also has a QuickWatch window that provides detailed information about a variable in the source code. To use this, simply right-click the variable you want to examine and select the QuickWatch menu option. In most cases, though, it is just as easy to use the standard Watch windows.

Watches are maintained between application executions. If you terminate an application and then rerun it, you don't have to add watches again — the IDE remembers what you were looking at the last time.

Stepping Through Code

So far, you've learned how to discover what is going on in your applications at the point where break mode is entered. Now it's time to see how you can use the IDE to step through code while remaining in break mode, which enables you to see the exact results of the code being executed. This is an extremely valuable technique for those of us who can't think as fast as computers can.

When break mode is entered, a cursor appears to the left of the code view (which may initially appear inside the red circle of a breakpoint if a breakpoint was used to enter break mode), by the line of code that is about to be executed, as shown in Figure 7-13.

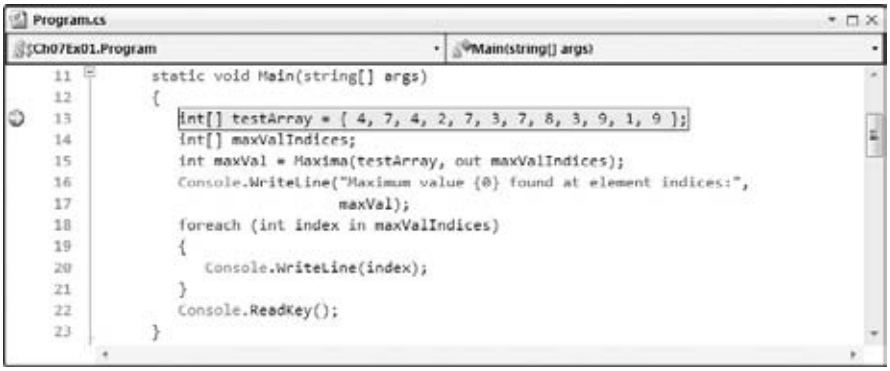

FIGURE 7-13

This shows you what point execution has reached when break mode is entered. At this point, you can have execution proceed on a line-by-line basis. To do so, you use some of the Debug toolbar buttons shown in Figure 7-14.

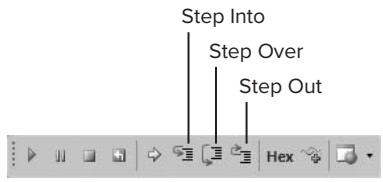

FIGURE 7-14

The sixth, seventh, and eighth icons control program flow in break mode. In order, they are as follows:

- **Step Into:** Execute and move to the next statement to execute.
- **Step Over:** Similar to Step Into, but won't enter nested blocks of code, including functions.
- **Step Out:** Run to the end of the code block and resume break mode at the statement that follows.

To look at every single operation carried out by the application, you can use Step Into to follow the instructions sequentially. This includes moving inside functions, such as `Maxima()` in the preceding example. Clicking this icon when the cursor reaches line 15, the call to `Maxima()`, results in the cursor moving to the first line inside the `Maxima()` function. Alternatively, clicking Step Over when you reach line 15 moves the cursor straight to line 16, without going through the code in `Maxima()` (although this code is still executed). If you do step into a function that you aren't interested in, you can click Step Out to return to the code that called the function. As you step through code, the values of variables are likely to change. If you keep an eye on the monitoring windows just discussed, you can clearly see this happening.

In code that has semantic errors, this technique may be the most useful one at your disposal. You can step through code right up to the point where you expect problems to occur, and the errors will be generated as if you were running the program normally. Along the way, watch the data to see just what is going wrong. Later in this chapter, you use this technique to find out what is happening in an example application.

Immediate and Command Windows

The Command (VS only) and Immediate windows (found on the Debug Windows menu) enable you to execute commands while an application is running. The Command window enables you to perform VS

operations manually (such as menu and toolbar operations), and the Immediate window enables you to execute additional code besides the source code lines being executed, and to evaluate expressions.

In VS, these windows are intrinsically linked (in fact, earlier versions of VS treated them as the same thing). You can even switch between them by entering commands: `immed` to move from the Command window to the Immediate window, and `>cmd` to move back.

This section concentrates on the Immediate window because the Command window is only really useful for complex operations and is only available in VS, whereas the Immediate window is available in both VS and VCE. The simplest use of this window is to evaluate expressions, a bit like a one-shot use of the Watch windows. To do this, type an expression and press Return. The information requested will then be displayed.
An example is shown in Figure 7-15.

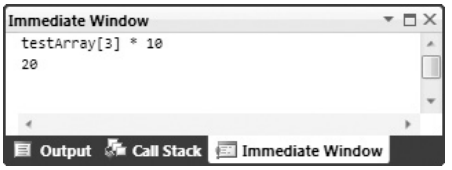

FIGURE 7-15

You can also change variable content here, as demonstrated in Figure 7-16.

In most cases, you can get the effects you want more easily using the variable monitoring windows shown earlier, but this technique is still handy for tweaking values, and it's good for testing expressions for which you are unlikely to be interested in the results later.

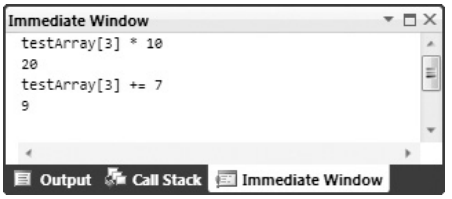

FIGURE 7-16

The Call Stack Window

The final window to look at is the Call Stack window, which shows you the way in which the current location was reached. In simple terms, this means showing the current function along with the function that called it, the function that called that, and so on (that is, a list of nested function calls). The exact points where calls are made are also recorded.

In the earlier example, entering break mode when in `Maxima()`, or moving into this function using code stepping, reveals the information shown in Figure 7-17.

FIGURE 7-17

If you double-click an entry, you are taken to the appropriate location, enabling you to track the way code execution has reached the current point. This window is particularly useful when errors are first detected, because you can see what happened immediately before the error. Where errors occur in commonly used functions, this helps you determine the source of the error.

NOTE *Sometimes the Call Stack window shows some very confusing information. For example, errors may occur outside of your applications due to using external functions in the wrong way. In such cases, this window could contain a long list of entries, but only one or two might look familiar. You can see external references (should you ever need to) by right-clicking in the window and selecting Show External Code.*

ERROR HANDLING

The first part of this chapter explained how to find and correct errors during application development so that they won't occur in release-level code. Sometimes, however, you know that errors are likely to occur and there is no way to be 100 percent sure that they won't. In those situations, it may be preferable to anticipate problems and write code that is robust enough to deal with these errors gracefully, without interrupting execution.

Error handling is the term for all techniques of this nature, and this section looks at exceptions and how you can deal with them. An exception is an error generated either in your code or in a function called by your code that occurs at runtime. The definition of error here is more vague than it has been up until now, because exceptions may be generated manually, in functions and so on. For example, you might generate an exception in a function if one of its string parameters doesn't start with the letter "a." Strictly speaking, this isn't an error outside of the context of the function, although it is treated as one by the code that calls the function.

You've seen exceptions a few times already in this book. Perhaps the simplest example is attempting to address an array element that is out of range:

```
int[] myArray = { 1, 2, 3, 4 };
int myElem = myArray[4];
```

This outputs the following exception message and then terminates the application:

```
Index was outside the bounds of the array.
```

NOTE *You've already seen some examples of the exception helper window that is displayed in previous chapters. It has a line connecting it to the offending code and includes links to reference topics in the .NET help files, as well as a View Detail link to more information about the exception.*

Exceptions are defined in namespaces, and most have names that make their purpose clear. In this example, the exception generated is called `System.IndexOutOfRangeException`, which makes sense because you have supplied an index that is not in the range of indices permissible in `myArray`. This message appears, and the application terminates, only when the exception is unhandled. In the next section, you'll see exactly what you have to do to handle an exception.

try . . . catch . . . finally

The C# language includes syntax for *structured exception handling* (SEH). Three keywords mark code as being able to handle exceptions, along with instructions specifying what to do when an exception occurs: `try`, `catch`, and `finally`. Each of these has an associated code block and must be used in consecutive lines of code. The basic structure is as follows:

```
try
{
   ...
}
catch (<exceptionType> e)
{
   ...
}
finally
{
   ...
}
```

It is also possible, however, to have a `try` block and a `finally` block with no `catch` block, or a `try` block with multiple `catch` blocks. If one or more `catch` blocks exist, then the `finally` block is optional; otherwise, it is mandatory. The usage of the blocks is as follows:

- `try` — Contains code that might throw exceptions ("throw" is the C# way of saying "generate" or "cause" when talking about exceptions)
- `catch` — Contains code to execute when exceptions are thrown. `catch` blocks may be set to respond only to specific exception types (such as `System.IndexOutOfRangeException`) using `<exceptionType>`, hence the ability to provide multiple `catch` blocks. It is also possible to omit this parameter entirely, to get a general `catch` block that responds to all exceptions.
- `finally` — Contains code that is always executed, either after the `try` block if no exception occurs, after a `catch` block if an exception is handled, or just before an unhandled exception moves "up the call stack." This phrase means that SEH allows you to nest `try...catch...finally` blocks inside each other, either directly or because of a call to a function within a `try` block. For example, if an exception isn't handled by any `catch` blocks in the called function, it might be handled by a `catch` block in the calling code. Eventually, if no `catch` blocks are matched, then the application will terminate. The fact that the `finally` block is processed before this happens is the reason for its existence; otherwise, you might just as well place code outside of the `try...catch...finally` structure. This nested functionality is discussed further in the "Notes on Exception Handling" section a little later, so don't worry if all that sounds a little confusing.

Here's the sequence of events that occurs after an exception occurs in code in a `try` block:

- The `try` block terminates at the point where the exception occurred.
- If a `catch` block exists, then a check is made to determine whether the block matches the type of exception that was thrown. If no `catch` block exists, then the `finally` block (which must be present if there are no `catch` blocks) executes.

- If a `catch` block exists but there is no match, then a check is made for other `catch` blocks.
- If a `catch` block matches the exception type, then the code it contains executes, and then the `finally` block executes if it is present.
- If no `catch` blocks match the exception type, then the `finally` block of code executes if it is present.

The following Try It Out demonstrates handling exceptions, throwing and handling them in several ways so you can see how things work.

TRY IT OUT Exception Handling

1. Create a new console application called Ch07Ex02 and save it in the directory `C:\BegVCSharp\Chapter07`.
2. Modify the code as follows (the line number comments shown here will help you match up your code to the discussion afterward, and they are duplicated in the downloadable code for this chapter for your convenience):


```
class Program
{
   static string[] eTypes = { "none", "simple", "index", "nested index" };

   static void Main(string[] args)
   {
      foreach (string eType in eTypes)
      {
         try
         {
            Console.WriteLine("Main() try block reached.");      // Line 23
            Console.WriteLine("ThrowException(\"{0}\") called.", eType);
                                                                 // Line 24
            ThrowException(eType);
            Console.WriteLine("Main() try block continues.");    // Line 26
         }
         catch (System.IndexOutOfRangeException e)               // Line 28
         {
            Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                              + " block reached. Message:\n\"{0}\"",
                              e.Message);
         }
         catch                                                   // Line 34
         {
            Console.WriteLine("Main() general catch block reached.");
         }
         finally
         {
            Console.WriteLine("Main() finally block reached.");
         }
         Console.WriteLine();
      }
      Console.ReadKey();
   }
```

```
static void ThrowException(string exceptionType)
{
                                                                    // Line 49
   Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
   switch (exceptionType)
   {
      case "none":
         Console.WriteLine("Not throwing an exception.");
         break;                                                     // Line 54
      case "simple":
         Console.WriteLine("Throwing System.Exception.");
         throw (new System.Exception());                            // Line 57
      case "index":
         Console.WriteLine("Throwing System.IndexOutOfRangeException.");
         eTypes[4] = "error";                                       // Line 60
         break;
      case "nested index":
         try                                                        // Line 63
         {
            Console.WriteLine("ThrowException(\"nested index\") " +
                              "try block reached.");
            Console.WriteLine("ThrowException(\"index\") called.");
            ThrowException("index");                                // Line 68
         }
         catch                                                      // Line 70
         {
            Console.WriteLine("ThrowException(\"nested index\") general"
                              + " catch block reached.");
         }
         finally
         {
            Console.WriteLine("ThrowException(\"nested index\") finally"
                              + " block reached.");
         }
         break;
   }
}
}
```

Code snippet Ch07Ex02\Program.cs

3. Run the application. The result is shown in Figure 7-18.

How It Works

This application has a `try` block in `Main()` that calls a function called `ThrowException()`. This function may throw exceptions, depending on the parameter it is called with:

- `ThrowException("none")`: Doesn't throw an exception
- `ThrowException("simple")`: Generates a general exception
- `ThrowException("index")`: Generates a `System.IndexOutOfRangeException` exception

- `ThrowException("nested index")`: Contains its own `try` block, which contains code that calls `ThrowException("index")` to generate a `System.IndexOutOfRangeException` exception

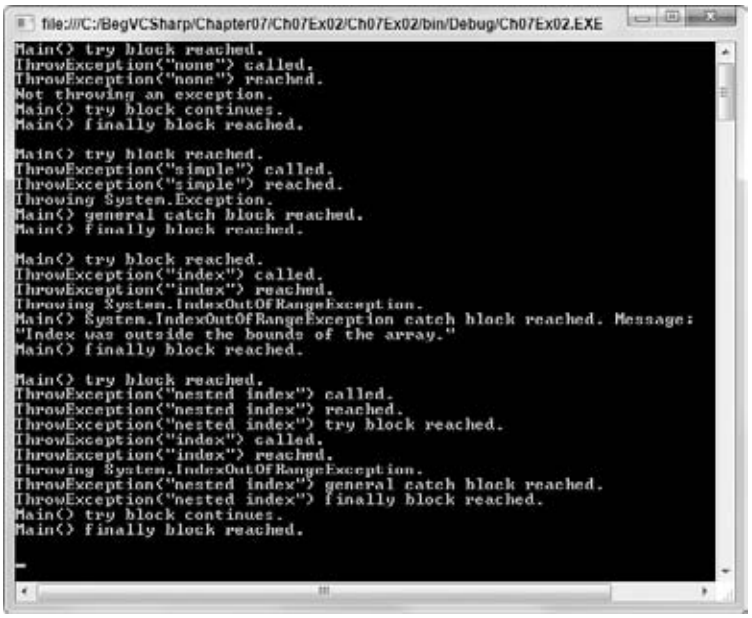

FIGURE 7-18

Each of these `string` parameters is held in the global `eTypes` array, which is iterated through in the `Main()` function to call `ThrowException()` once with each possible parameter. During this iteration, various messages are written to the console to indicate what is happening. This code gives you an excellent opportunity to use the code-stepping techniques shown earlier in the chapter. By working your way through the code one line at a time, you can see exactly how code execution progresses.

Add a new breakpoint (with the default properties) to line 23 of the code, which reads as follows:

```
Console.WriteLine("Main() try block reached.");
```

NOTE *Code is referred to by line numbers as they appear in the downloadable version of this code. If you have line numbers turned off, remember that you can turn them back on (select Tools ➪ Options in the Text Editor ➪ C# ➪ General options section). Comments are included in the preceding code so that you can follow the text without having the file open in front of you.*

Run the application in debug mode. Almost immediately, the program will enter break mode, with the cursor on line 23. If you select the Locals tab in the variable monitoring window, you should see that `eType` is currently `"none"`. Use the Step Into button to process lines 23 and 24, and confirm that the first line of

text has been written to the console. Next, use the Step Into button to step into the `ThrowException()` function on line 25.

Once in the `ThrowException()` function (on line 49), the Locals window changes. `eType` and `args` are no longer in scope (they are local to `Main()`); instead, you see the local `exceptionType` argument, which is, of course, `"none"`. Keep pressing Step Into and you'll reach the `switch` statement that checks the value of `exceptionType` and executes the code that writes out the string `Not throwing an exception` to the screen. When you execute the `break` statement (on line 54), you exit the function and resume processing in `Main()` at line 26. Because no exception was thrown, the `try` block continues.

Next, processing continues with the `finally` block. Click Step Into a few more times to complete the `finally` block and the first cycle of the `foreach` loop. The next time you reach line 25, `ThrowException()` is called using a different parameter, `"simple"`.

Continue using Step Into through `ThrowException()`, and you'll eventually reach line 57:

```
throw (new System.Exception());
```

You use the C# `throw` keyword to generate an exception. This keyword simply needs to be provided with a `new`-initialized exception as a parameter, and it will throw that exception. Here, you are using another exception from the `System` namespace, `System.Exception`.

NOTE *When you use* `throw` *in a* `case` *block, no* `break;` *statement is necessary.* `throw` *is enough to end execution of the block.*

When you process this statement with Step Into, you find yourself at the general `catch` block starting on line 34. There was no match with the earlier `catch` block starting on line 28, so this one is processed instead. Stepping through this code takes you through this block, through the `finally` block, and back into another loop cycle that calls `ThrowException()` with a new parameter on line 25. This time the parameter is `"index"`.

Now `ThrowException()` generates an exception on line 60:

```
eTypes[4] = "error";
```

The `eTypes` array is global, so you have access to it here. However, here you are attempting to access the fifth element in the array (remember that counting starts at 0), which generates a `System.IndexOutOfRangeException` exception.

This time there is a matched `catch` block in `Main()`, and stepping into the code takes you to this block, starting at line 28. The `Console.WriteLine()` call in this block writes out the message stored in the exception using `e.Message` (you have access to the exception through the parameter of the `catch` block). Again, stepping through takes you through the `finally` block (but not the second `catch` block, as the exception is already handled) and back into the loop cycle, again calling `ThrowException()` on line 25.

When you reach the `switch` structure in `ThrowException()`, this time you enter a new `try` block, starting on line 63. When you reach line 68, you perform a nested call to `ThrowException()`, this time with the parameter `"index"`. You can use the Step Over button to skip the lines of code that are executed here because you've been through them already. As before, this call generates a `System.IndexOutOfRangeException`

exception, but this time it's handled in the nested `try...catch...finally` structure, the one in `ThrowException()`. This structure has no explicit match for this type of exception, so the general `catch` block (starting on line 70) deals with it.

As with the earlier exception handling, you now step through this `catch` block and the associated `finally` block, and reach the end of the function call, but with one crucial difference: Although an exception was thrown, it was also handled — by the code in `ThrowException()`. This means there is no exception left to handle in `Main()`, so you go straight to the `finally` block, and then the application terminates.

Listing and Configuring Exceptions

The .NET Framework contains a whole host of exception types, and you are free to throw and handle any of these in your own code, or even throw them from your code so that they may be caught in more complex applications. The IDE supplies a dialog for examining and editing the available exceptions, which can be called up with the Debug ➪ Exceptions menu item (or by pressing Ctrl+D, E). Figure 7-19 shows the dialog (the list will vary if you use VCE, which only includes the second and third entries shown in Figure 7-19).

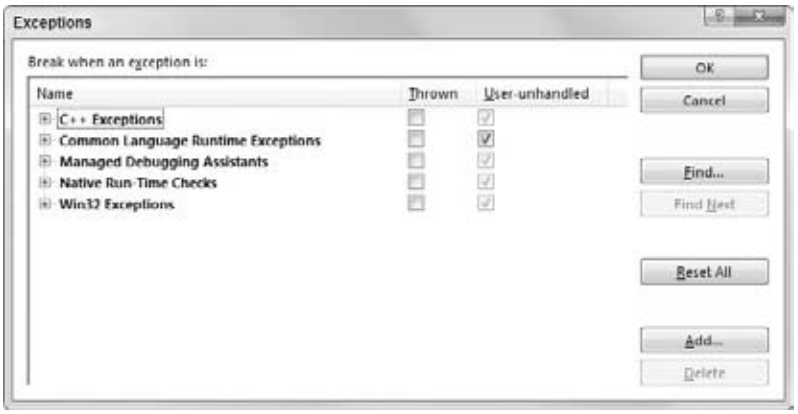

FIGURE 7-19

Exceptions are listed by category and .NET library namespace. You can see the exceptions in the `System` namespace by expanding the Common Language Runtime Exceptions tab, and then the System tab. The list includes the `System.IndexOutOfRangeException` exception you used earlier.

Each exception may be configured using the check boxes shown. You can use the first option, (break when) `Thrown`, to cause a break into the debugger even for exceptions that are handled. The second option enables you to ignore unhandled exceptions, and suffer the consequences. In most cases, this results in break mode being entered, so you will likely need to do this only in exceptional circumstances.

Typically, the default settings here are fine.

Notes on Exception Handling

You must always supply `catch` blocks for more specific exceptions before more general catching. If you get this wrong, the application will fail to compile. Note also that you can throw exceptions from within `catch` blocks, either in the ways used in the previous example or simply by using the following expression:

```
throw;
```

This expression results in the exception handled by the `catch` block being rethrown. If you throw an exception in this way, it will not be handled by the current `try...catch...finally` block, but by parent code (although the `finally` block in the nested structure will still execute).

For example, if you changed the `try...catch...finally` block in `ThrowException()` as follows:

```
try
{
   Console.WriteLine("ThrowException(\"nested index\") " +
                     "try block reached.");
   Console.WriteLine("ThrowException(\"index\") called.");
   ThrowException("index");
}
catch
{
   Console.WriteLine("ThrowException(\"nested index\") general"
      + " catch block reached.");
   throw;
}
finally
{
   Console.WriteLine("ThrowException(\"nested index\") finally"
      + " block reached.");
}
```

then execution would proceed first to the `finally` block shown here, then with the matching `catch` block in `Main()`. The resulting console output changes, as shown in Figure 7-20.

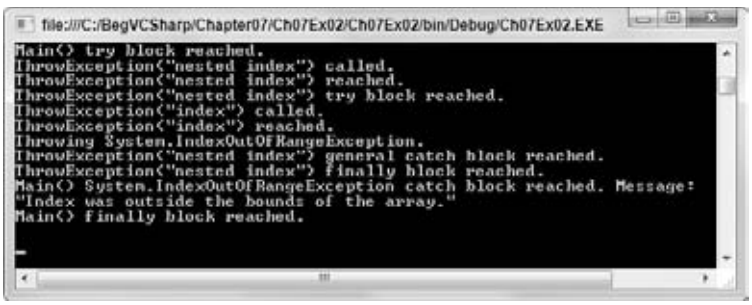

FIGURE 7-20

This screenshot shows extra lines of output from the `Main()` function, as the `System.IndexOutOfRangeException` is caught in this function.

SUMMARY

This chapter concentrates on techniques that you can use to debug your applications. A variety of techniques are possible, most of which are available for whatever type of project you are creating, not just console applications.

You have now covered everything that you need to produce simple console applications, along with the methods for debugging them. From the next chapter onward, you'll look at the powerful technique of object-oriented programming.

EXERCISES

1. "Using `Trace.WriteLine()` is preferable to using `Debug.WriteLine()`, as the `Debug` version only works in debug builds." Do you agree with this statement? If so, why?

2. Provide code for a simple application containing a loop that generates an error after 5,000 cycles. Use a breakpoint to enter break mode just before the error is caused on the 5000th cycle. (Note: A simple way to generate an error is to attempt to access a nonexistent array element, such as `myArray[1000]` in an array with 100 elements.)

3. "`finally` code blocks only execute if a `catch` block isn't executed." True or false?

4. Given the enumeration data type `orientation` defined in the following code, write an application that uses structured exception handling (SEH) to cast a `byte`-type variable into an `orientation`-type variable in a safe way. (Note: You can force exceptions to be thrown using the `checked` keyword, an example of which is shown here. This code should be used in your application.)

```
enum Orientation : byte
{
   North = 1,
   South = 2,
   East  = 3,
   West  = 4
}
myDirection = checked((Orientation)myByte);
```

Answers to Exercises can be found in Appendix A.

► WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Error types	Fatal errors cause your application to fail completely, either at compile time (syntax errors) or at runtime. Semantic, or logic, errors are more insidious, and may cause your application to function incorrectly or unpredictably.
Outputting debugging information	You can write code that outputs helpful information to the Output window to aid debugging in the IDE. You do this with the `Debug` and `Trace` family of functions, where `Debug` functions are ignored in release builds. For production applications, you may want to write debugging output to a log file instead. In VS, you can also use tracepoints to output debugging information.
Break mode	You can enter break mode (essentially a state where the application is paused) manually, through breakpoints, through assertions, or when unhandled exceptions occur. You can add breakpoints anywhere in your code, and in VS you can configure breakpoints to only break execution under specific conditions. When in break mode, you can inspect the content of variables (with the help of various debug information windows) and step through code a line at a time to assist you in determining where errors may be occurring.
Exceptions	Exceptions are errors that occur at runtime and that you can trap and process programmatically to prevent your application from terminating. There are many different types of exceptions that might occur when you call functions or manipulate variables. You can also generate exceptions with the `throw` keyword.
Exception handling	Exceptions that are not handled in your code will cause the application to terminate. You handle exceptions with `try`, `catch`, and `finally` code blocks. `try` blocks mark out a section of code for which exception handling is enabled. `catch` blocks consist of code that is executed only if an exception occurs, and can match specific types of exceptions. You can include multiple `catch` blocks. `finally` blocks specify code that is executed after exception handling has occurred, or after the `try` block finishes if no exception occurs. You can include only a single `finally` block, and if you include any `catch` blocks, then the `finally` block is optional.

8

Introduction to Object-Oriented Programming

WHAT YOU WILL LEARN IN THIS CHAPTER

- What object-oriented programming is
- OOP techniques
- How Windows Forms applications rely on OOP

At this point in the book, you've covered all the basics of C# syntax and programming, and have learned how to debug your applications. Already, you can assemble usable console applications. However, to access the real power of the C# language and the .NET Framework, you need to make use of *object-oriented programming* (OOP) techniques. In fact, as you will soon see, you've been using these techniques already, though to keep things simple we haven't focused on this.

This chapter steers away from code temporarily and focuses instead on the principles behind OOP. This leads you back into the C# language because it has a symbiotic relationship with OOP. All of the concepts introduced in this chapter are revisited in later chapters, with illustrative code — so don't panic if you don't grasp everything in the first read-through of this material.

To start with, you'll look at the basics of OOP, which include answering that most fundamental of questions, "What is an *object?*" You will quickly find that a lot of terminology related to OOP can be quite confusing at first, but plenty of explanations are provided. You will also see that using OOP requires you to look at programming in a different way.

As well as discussing the general principles of OOP, this chapter also looks at an area requiring a thorough understanding of OOP: Windows Forms applications. This type of application (which makes use of the Windows environment, with features such as menus, buttons, and so on) provides plenty of scope for description, and you will be able to observe OOP points effectively in the Windows Forms environment.

NOTE *OOP as presented in this chapter is really .NET OOP, and some of the techniques presented here don't apply to other OOP environments. When programming in C#, you use .NET-specific OOP, so it makes sense to concentrate on these aspects.*

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming is a relatively new approach to creating computer applications that seeks to address many of the problems with traditional programming techniques. The type of programming you have seen so far is known as *functional* (or *procedural*) programming, often resulting in so-called monolithic applications, meaning all functionality is contained in a few modules of code (often just one). With OOP techniques, you often use many more modules of code, each offering specific functionality, and each module may be isolated or even completely independent of the others. This modular method of programming gives you much more versatility and provides more opportunity for code reuse.

To illustrate this further, imagine that a high-performance application on your computer is a top-of-the-range race car. Written with traditional programming techniques, this sports car is basically a single unit. If you want to improve this car, then you have to replace the whole unit by sending it back to the manufacturer and getting their expert mechanics to upgrade it, or by buying a new one. If OOP techniques are used, however, you can simply buy a new engine from the manufacturer and follow their instructions to replace it yourself, rather than taking a hacksaw to the bodywork.

In a more traditional application, the flow of execution is often simple and linear. Applications are loaded into memory, begin executing at point A, end at point B, and are then unloaded from memory. Along the way various other entities might be used, such as files on storage media, or the capabilities of a video card, but the main body of the processing occurs in one place. The code along the way is generally concerned with manipulating data through various mathematical and logical means. The methods of manipulation are usually quite simple, using basic types such as integers and Boolean values to build more complex representations of data.

With OOP, things are rarely so linear. Although the same results are achieved, the way of getting there is often very different. OOP techniques are firmly rooted in the structure and meaning of data, and the interaction between that data and other data. This usually means putting more effort into the design stages of a project, but it has the benefit of extensibility. After an agreement is made as to the representation of a specific type of data, that agreement can be worked into later versions of an application, and even entirely new applications. The fact that such an agreement exists can reduce development time dramatically. This explains how the race car example works. The agreement here is how the code for the "engine" is structured, such that new code (for a new engine) can be substituted with ease, rather than requiring a trip back to the manufacturer. It also means that the engine, once created, can be used for other purposes. You could put it in a different car, or use it to power a submarine, for example.

OOP programming often simplifies things by providing an agreement about the approach to data representation as well as about the structure and usage of more abstract entities. For example, an agreement can be made not just on the format of data that should be used to send output to a device such as a printer, but also on the methods of data exchange with that device, including what instructions it

understands, and so on. In the race car analogy, the agreement would include how the engine connects to the fuel tank, how it passes drive power to the wheels, and so on.

As the name of the technology suggests, this is achieved using *objects*.

What Is an Object?

An *object* is a building block of an OOP application. This building block encapsulates part of the application, which may be a process, a chunk of data, or a more abstract entity.

In the simplest sense, an object may be very similar to a struct type such as those shown earlier in the book, containing members of variable and function types. The variables contained make up the data stored in the object, and the functions contained allow access to the object's functionality. Slightly more complex objects might not maintain any data; instead, they can represent a process by containing only functions. For example, an object representing a printer might be used, which would have functions enabling control over a printer (so you can print a document, a test page, and so on).

Objects in C# are created from types, just like the variables you've seen already. The type of an object is known by a special name in OOP, its *class*. You can use class definitions to *instantiate* objects, which means creating a real, named *instance* of a class. The phrases *instance of a class* and *object* mean the same thing here; but *class* and *object* mean fundamentally different things.

NOTE *The terms* class *and* object *are often confused, and it is important to understand the distinction. It may help to visualize these terms using the earlier race car analogy. Think of a class as the template for the car, or perhaps the plans used to build the car. The car itself is an instance of those plans, so it could be referred to as an object.*

In this chapter, you work with classes and objects using *Unified Modeling Language* (UML) syntax. UML is designed for modeling applications, from the objects that build them to the operations they perform to the use cases that are expected. Here, you use only the basics of this language, which are explained as you go along. UML is a specialized subject to which entire books are devoted, so its more complex aspects are not covered here.

NOTE *VS has a class viewer that is a powerful tool in its own right that can be used to display classes in a similar way. For simplicity, though, the figures in this chapter were hand drawn.*

Figure 8-1 shows a UML representation of your printer class, called `Printer`. The class name is shown in the top section of this box (you learn about the bottom two sections a little later).

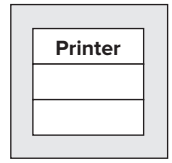

FIGURE 8-1

Figure 8-2 shows a UML representation of an instance of this `Printer` class called `myPrinter`.

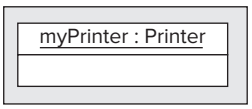

FIGURE 8-2

Here, the instance name is shown first in the top section, followed by the name of its class. The two names are separated by a colon.

Properties and Fields

Properties and fields provide access to the data contained in an object. This object data is what differentiates separate objects because it is possible for different objects of the same class to have different values stored in properties and fields.

The various pieces of data contained in an object together make up the *state* of that object. Imagine an object class that represents a cup of coffee, called `CupOfCoffee`. When you instantiate this class (that is, create an object of this class), you must provide it with a state for it to be meaningful. In this case, you might use properties and fields to enable the code that uses this object to set the type of coffee used, whether the coffee contains milk and/or sugar, whether the coffee is instant, and so on. A given coffee cup object would then have a given state, such as "Columbian filter coffee with milk and two sugars."

Both fields and properties are typed, so you can store information in them as `string` values, as `int` values, and so on. However, properties differ from fields in that they don't provide direct access to data. Objects can shield users from the nitty-gritty details of their data, which needn't be represented on a one-to-one basis in the properties that exist. If you used a field for the number of sugars in a `CupOfCoffee` instance, then users could place whatever values they liked in the field, limited only by the limits of the type used to store this information. If, for example, you used an `int` to store this data, then users could use any value between −2147483648 and 2147483647, as shown in Chapter 3. Obviously, not all values make sense, particularly the negative ones, and some of the large positive amounts might require an inordinately large cup. If you use a property for this information, then you could limit this value to, say, a number between 0 and 2.

In general, it is better to provide properties rather than fields for state access because you have more control over various behaviors. This choice doesn't affect code that uses object instances because the syntax for using properties and fields is the same.

Read/write access to properties may also be clearly defined by an object. Certain properties may be read-only, allowing you to see what they are but not change them (at least not directly). This is often a useful technique for reading several pieces of state simultaneously. You might have a read-only property of the `CupOfCoffee` class called `Description`, returning a string representing the state of an instance of this class (such as the string given earlier) when requested. You might be able to assemble the same data by interrogating several properties, but a property such as this one may save you time and effort. You might also have write-only properties that operate in a similar way.

As well as this read/write access for properties, you can also specify a different sort of access permission for both fields and properties, known as *accessibility*. Accessibility determines which code can access these members — that is, whether they are available to all code (public), only to code within the class (private), or should use a more complex scheme (covered in more detail later in the chapter, when it becomes pertinent). One common practice is to make fields private and provide access to them via public properties. This means that code within the class has direct access to data stored in the field, while the public property shields external users from this data and prevents them from placing invalid content there. Public members are said to be *exposed* by the class.

One way to visualize this is to equate it with variable scope. Private fields and properties, for example, can be thought of as local to the object that possesses them, whereas the scope of public fields and properties also encompasses code external to the object.

In the UML representation of a class, you use the second section to display properties and fields, as shown in Figure 8-3.

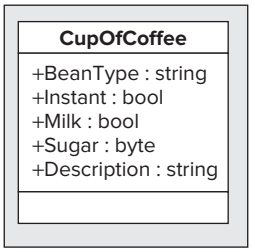

FIGURE 8-3

This is a representation of the `CupOfCoffee` class, with five members (properties or fields, because no distinction is made in UML) defined as discussed earlier. Each of the entries contains the following information:

- Accessibility: A + symbol is used for a public member, a − symbol is used for a private member. In general, though, private members are not shown in the diagrams in this chapter because this information is internal to the class. No information is provided as to read/write access.
- The member name.
- The type of the member.

A colon is used to separate the member names and types.

Methods

Method is the term used to refer to functions exposed by objects. These may be called in the same way as any other function and may use return values and parameters in the same way — you looked at functions in detail in Chapter 6.

Methods are used to provide access to the object's functionality. Like fields and properties, they can be public or private, restricting access to external code as necessary. They often make use of an object's state to affect their operations, and have access to private members, such as private fields, if required. For example, the `CupOfCoffee` class might define a method called `AddSugar()`, which would provide a more readable syntax for incrementing the amount of sugar than setting the corresponding `Sugar` property.

In UML, class boxes show methods in the third section, as shown in Figure 8-4.

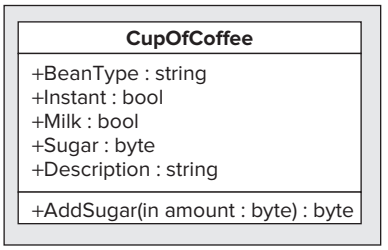

FIGURE 8-4

The syntax here is similar to that for fields and properties, except that the type shown at the end is the return type, and method parameters are shown. Each parameter is displayed in UML with one of the following identifiers: `in`, `out`, or `inout`. These are used to signify the direction of data flow, where `out` and `inout` roughly correspond to the use of the C# keywords `out` and `ref` described in Chapter 6. `in` roughly corresponds to the default C# behavior, where neither the `out` nor `ref` keyword is used.

Everything's an Object

At this point, it's time to come clean: You have been using objects, properties, and methods throughout this book. In fact, everything in C# and the .NET Framework is an object! The `Main()` function in a

console application is a method of a class. Every variable type you've looked at is a class. Every command you have used has been a property or a method, such as `<String>.Length`, `<String>.ToUpper()`, and so on. (The period character here separates the object instance's name from the property or method name, and methods are shown with `()` at the end to differentiate them from properties.)

Objects really are everywhere, and the syntax to use them is often very simple. It has certainly been simple enough for you to concentrate on some of the more fundamental aspects of C# up until now. From this point on, you'll begin to look at objects in detail. Bear in mind that the concepts introduced here have far-reaching consequences — applying even to that simple little `int` variable you've been happily playing around with.

The Life Cycle of an Object

Every object has a clearly defined life cycle. Apart from the normal state of ''being in use,'' this life cycle includes two important stages:

- **Construction:** When an object is first instantiated it needs to be initialized. This initialization is known as *construction* and is carried out by a constructor function, often referred to simply as a *constructor* for convenience.
- **Destruction:** When an object is destroyed, there are often some clean-up tasks to perform, such as freeing memory. This is the job of a destructor function, also known as a *destructor*.

Constructors

Basic initialization of an object is automatic. For example, you don't have to worry about finding the memory to fit a new object into. However, at times you will want to perform additional tasks during an object's initialization stage, such as initializing the data stored by an object. A constructor is what you use to do this.

All class definitions contain at least one constructor. These constructors may include a *default constructor,* which is a parameterless method with the same name as the class itself. A class definition might also include several constructor methods with parameters, known as *nondefault constructors*. These enable code that instantiates an object to do so in many ways, perhaps providing initial values for data stored in the object.

In C#, constructors are called using the `new` keyword. For example, you could instantiate a `CupOfCoffee` object using its default constructor in the following way:

```
CupOfCoffee myCup = new CupOfCoffee();
```

Objects may also be instantiated using nondefault constructors. For example, the `CupOfCoffee` class might have a nondefault constructor that uses a parameter to set the bean type at instantiation:

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

Constructors, like fields, properties, and methods, may be public or private. Code external to a class can't instantiate an object using a private constructor; it must use a public constructor. In this way, you can, for example, force users of your classes to use a nondefault constructor (by making the default constructor private).

Some classes have no public constructors, meaning it is impossible for external code to instantiate them (they are said to be *noncreatable*). However, that doesn't make them completely useless, as you will see shortly.

Destructors

Destructors are used by the .NET Framework to clean up after objects. In general, you don't have to provide code for a destructor method; instead, the default operation does the work for you. However, you can provide specific instructions if anything important needs to be done before the object instance is deleted.

When a variable goes out of scope, for example, it may not be accessible from your code, but it may still exist somewhere in your computer's memory. Only when the .NET runtime performs its garbage collection cleanup is the instance completely destroyed.

NOTE *Don't rely on the destructor to free up resources used by an object instance, as this may occur long after the object is of no further use to you. If the resources in use are critical, then this can cause problems. However, there is a solution to this — described in "Disposable Objects" later in this chapter.*

Static and Instance Class Members

As well as having members such as properties, methods, and fields that are specific to object instances, it is also possible to have *static* (also known as *shared*, particularly to our Visual Basic brethren) members, which may be methods, properties, or fields. Static members are shared between instances of a class, so they can be thought of as global for objects of a given class. Static properties and fields enable you to access data that is independent of any object instances, and static methods enable you to execute commands related to the class type but not specific to object instances. When using static members, in fact, you don't even need to instantiate an object.

For example, the `Console.WriteLine()` and `Convert.ToString()` methods you have been using are static. At no point do you need to instantiate the `Console` or `Convert` classes (indeed, if you try, you'll find that you can't, as the constructors of these classes aren't publicly accessible, as discussed earlier).

There are many situations such as these where static properties and methods can be used to good effect. For example, you might use a static property to keep track of how many instances of a class have been created. In UML syntax, static members of classes appear with underlining, as shown in Figure 8-5.

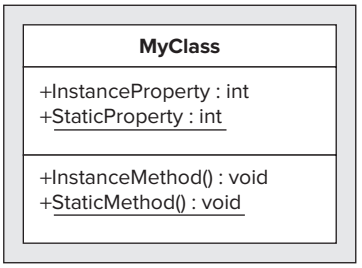

FIGURE 8-5

Static Constructors

When using static members in a class, you may want to initialize these members beforehand. You can supply a static member with an initial value as part of its declaration, but sometimes you may want to

perform a more complex initialization, or perhaps perform some operations before assigning values or allowing static methods to execute.

You can use a static constructor to perform initialization tasks of this type. A class can have a single static constructor, which must have no access modifiers and cannot have any parameters. A static constructor can never be called directly; instead, it is executed when one of the following occurs:

- An instance of the class containing the static constructor is created.
- A static member of the class containing the static constructor is accessed.

In both cases, the static constructor is called first, before the class is instantiated or static members accessed. No matter how many instances of a class are created, its static constructor will only be called once. To differentiate between static constructors and the constructors described earlier in this chapter, all nonstatic constructors are also known as *instance constructors*.

Static Classes

Often, you will want to use classes that contain only static members and cannot be used to instantiate objects (such as `Console`). A shorthand way to do this, rather than make the constructors of the class private, is to use a *static class*. A static class can contain only static members and can't have instance constructors, since by implication it can never be instantiated. Static classes can, however, have a static constructor, as described in the preceding section.

NOTE *If you are completely new to OOP, you might like to take a break before embarking on the remainder of this chapter. It is important to fully grasp the fundamentals before learning about the more complicated aspects of this methodology.*

OOP TECHNIQUES

Now that you know the basics, and what objects are and how they work, spend some time looking at some of the other features of objects. This section covers all of the following:

- Interfaces
- Inheritance
- Polymorphism
- Relationships between objects
- Operator overloading
- Events
- Reference versus value types

Interfaces

An interface is a collection of public instance (that is, nonstatic) methods and properties that are grouped together to encapsulate specific functionality. After an interface has been defined, you can implement it in a class. This means that the class will then support all of the properties and members specified by the interface.

Interfaces cannot exist on their own. You can't "instantiate an interface" as you can a class. In addition, interfaces cannot contain any code that implements its members; it just defines the members themselves. The implementation must come from classes that implement the interface.

In the earlier coffee example, you might group together many of the more general-purpose properties and methods into an interface, such as `AddSugar()`, `Milk`, `Sugar`, and `Instant`. You could call this interface something like `IHotDrink` (interface names are normally prefixed with a capital `I`). You could use this interface on other objects, perhaps those of a `CupOfTea` class. You could therefore treat these objects in a similar way, and they may still have their own individual properties (`BeanType` for `CupOfCoffee` and `LeafType` for `CupOfTea`, for example).

Interfaces implemented on objects in UML are shown using a *lollipop* syntax. In Figure 8-6, members of `IHotDrink` are split into a separate box using class-like syntax.

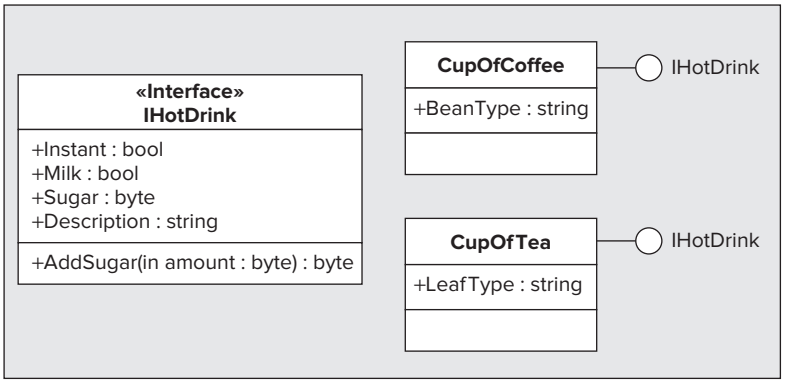

FIGURE 8-6

A class can support multiple interfaces, and multiple classes can support the same interface. The concept of an interface, therefore, makes life easier for users and other developers. For example, you might have some code that uses an object with a certain interface. Provided that you don't use other properties and methods of this object, it is possible to replace one object with another (code using the `IHotDrink` interface shown earlier could work with both `CupOfCoffee` and `CupOfTea` instances, for example). In addition, the developer of the object itself could supply you with an updated version of an object, and as long as it supports an interface already in use it would be easy to use this new version in your code.

Once an interface is published — that is, it has been made available to other developers or end users — it is good practice not to change it. One way of thinking about this is to imagine the interface as a contract between class creators and class consumers. You are effectively saying, "Every class that supports interface `X` will support these methods and properties." If the interface changes later,

perhaps due to an upgrade of the underlying code, this could cause consumers of that interface to run it incorrectly, or even fail. Instead, you should create a new interface that extends the old one, perhaps including a version number, such as `X2`. This has become the standard way of doing things, and you are likely to come across numbered interfaces frequently.

Disposable Objects

One interface of particular interest is `IDisposable`. An object that supports the `IDisposable` interface must implement the `Dispose()` method — that is, it must provide code for this method. This method can be called when an object is no longer needed (just before it goes out of scope, for example) and should be used to free up any critical resources that might otherwise linger until the destructor method is called on garbage collection. This gives you more control over the resources used by your objects.

C# enables you to use a structure that makes excellent use of this method. The `using` keyword enables you to initialize an object that uses critical resources in a code block, where `Dispose()` is automatically called at the end of the code block:

```
<ClassName> <VariableName> = new <ClassName>();

...


using (<VariableName>)
{
   ...
}
```

Alternatively, you can instantiate the object `<VariableName>` as part of the `using` statement:

```
using (<ClassName> <VariableName> = new <ClassName>())
{
   ...
}
```

In both cases, the variable `<VariableName>` will be usable within the `using` code block and will be disposed of automatically at the end (that is, `Dispose()` is called when the code block finishes executing).

Inheritance

Inheritance is one of the most important features of OOP. Any class may *inherit* from another, which means that it will have all the members of the class from which it inherits. In OOP terminology, the class being inherited from (*derived* from) is the *parent* class (also known as the *base* class). Classes in C# may derive only from a single base class directly, although of course that base class may have a base class of its own, and so on.

Inheritance enables you to extend or create more specific classes from a single, more generic base class. For example, consider a class that represents a farm animal (as used by ace octogenarian developer Old MacDonald in his livestock application). This class might be called `Animal` and possess methods such as `EatFood()` or `Breed()`. You could create a derived class called `Cow`, which would support all of these methods but might also supply its own, such as `Moo()` and `SupplyMilk()`. You could also create another derived class, `Chicken`, with `Cluck()` and `LayEgg()` methods.

In UML, you indicate inheritance using arrows, as shown in Figure 8-7.

NOTE *In Figure 8-7, the member return types are omitted for clarity.*

When using inheritance from a base class, the question of member accessibility becomes an important one. Private members of the base class are not accessible from a derived class, but public members are. However, public members are accessible to both the derived class and external code. Therefore, if you could use only these two levels of accessibility, you couldn't have a member that was accessible both by the base class and the derived class but not external code.

To get around this, there is a third type of accessibility, *protected,* in which only derived classes have access to a member. As far as external code is aware, this is identical to a private member — it doesn't have access in either case.

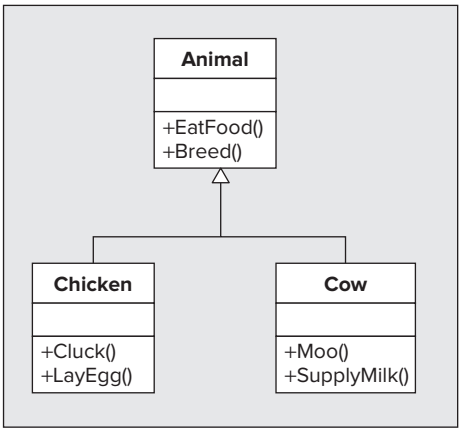

FIGURE 8-7

As well as defining the protection level of a member, you can also define an inheritance behavior for it. Members of a base class may be *virtual,* which means that the member can be overridden by the class that inherits it. Therefore, the derived class may provide an alternative implementation for the member. This alternative implementation doesn't delete the original code, which is still accessible from within the class, but it does shield it from external code. If no alternative is supplied, then any external code that uses the member through the derived class automatically uses the base class implementation of the member.

NOTE *Virtual members cannot be private because that would cause a paradox — it is impossible to say that a member can be overridden by a derived class at the same time you say that it is inaccessible from the derived class.*

In the animals example, you could make `EatFood()` virtual and provide a new implementation for it on any derived class — for example, just on the `Cow` class, as shown in Figure 8-8. This displays the `EatFood()` method on the `Animal` and `Cow` classes to signify that they have their own implementations.

Base classes may also be defined as *abstract* classes. An abstract class can't be instantiated directly; to use it you need to inherit from it. Abstract classes may have abstract members, which have no implementation in the base class, so an implementation must be supplied in the derived class. If `Animal` were an abstract class, then the UML would look as shown in Figure 8-9.

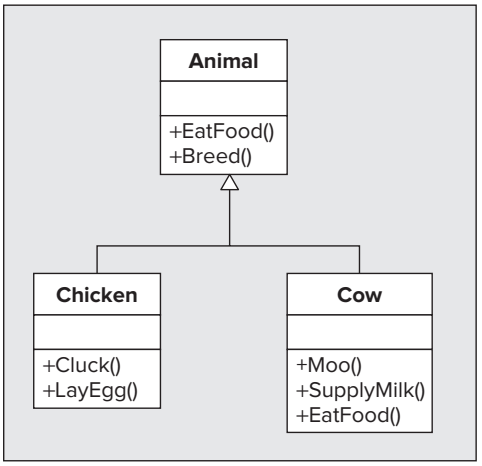

FIGURE 8-8

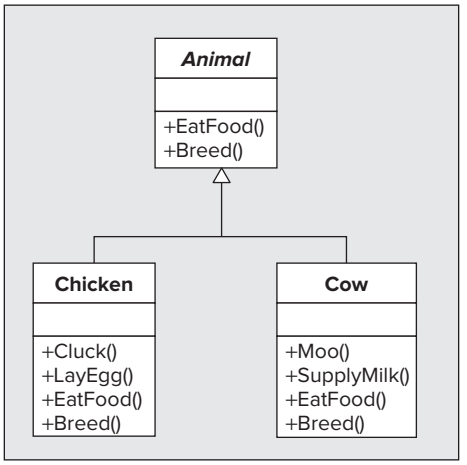

FIGURE 8-9

NOTE *Abstract class names are shown in italics (or with a dashed line for their boxes).*

In Figure 8-9, both `EatFood()` and `Breed()` are shown in the derived classes `Chicken` and `Cow`, implying that these methods are either abstract (and, therefore, must be overridden in derived classes) or virtual (and, in this case, have been overridden in `Chicken` and `Cow`). Of course, abstract base classes can provide implementation of members, which is very common. The fact that you can't instantiate an abstract class doesn't mean you can't encapsulate functionality in it.

Finally, a class may be *sealed*. A sealed class may not be used as a base class, so no derived classes are possible.

C# provides a common base class for all objects called `object` (which is an alias for the `System.Object` class in the .NET Framework). You take a closer look at this class in Chapter 9.

NOTE *Interfaces, described earlier in this chapter, may also inherit from other interfaces. Unlike classes, interfaces may inherit from multiple base interfaces (in the same way that classes can support multiple interfaces).*

Polymorphism

One consequence of inheritance is that classes deriving from a base class have an overlap in the methods and properties that they expose. Because of this, it is often possible to treat objects instantiated from classes with a base type in common using identical syntax. For example, if a base class called `Animal` has a method called `EatFood()`, then the syntax for calling this method from the derived classes `Cow` and `Chicken` will be similar:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

Polymorphism takes this a step further. You can assign a variable that is of a derived type to a variable of one the base types, as shown here:

```
Animal myAnimal = myCow;
```

No casting is required for this. You can then call methods of the base class through this variable:

```
myAnimal.EatFood();
```

This results in the implementation of `EatFood()` in the derived class being called. Note that you can't call methods defined on the derived class in the same way. The following code won't work:

```
myAnimal.Moo();
```

However, you can cast a base type variable into a derived class variable and call the method of the derived class that way:

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

This casting causes an exception to be raised if the type of the original variable was anything other than `Cow` or a class derived from `Cow`. There are ways to determine the type of an object, which you'll learn in the next chapter.

Polymorphism is an extremely useful technique for performing tasks with a minimum of code on different objects descending from a single class. It isn't just classes sharing the same parent class that can make use of polymorphism. It is also possible to treat, say, a child and a grandchild class in the same way, as long as there is a common class in their inheritance hierarchy.

As a further note here, remember that in C# all classes derive from the base class `object` at the root of their inheritance hierarchies. It is therefore possible to treat all objects as instances of the class `object`. This is how `Console.WriteLine()` is able to process an almost infinite number of parameter combinations when building strings. Every parameter after the first is treated as an `object` instance, allowing output from any object to be written to the screen. To do this, the method `ToString()` (a member of `object`) is called. You can override this method to provide an implementation suitable for your class, or simply use the default, which returns the class name (qualified according to any namespaces it is in).

Interface Polymorphism

Although you can't instantiate interfaces in the same way as objects, you can have a variable of an interface type. You can then use the variable to access methods and properties exposed by this interface on objects that support it.

For example, suppose that instead of an `Animal` base class being used to supply the `EatFood()` method, you place this `EatFood()` method on an interface called `IConsume`. The `Cow` and `Chicken` classes could both support this interface, the only difference being that they are forced to provide an implementation

for `EatFood()` because interfaces contain no implementation. You can then access this method using code such as the following:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

This provides a simple way for multiple objects to be called in the same manner, and it doesn't rely on a common base class. For example, this interface could be implemented by a class called `VenusFlyTrap` that derives from `Vegetable` instead of `Animal`:

```
VenusFlyTrap myVenusFlyTrap = new VenusFlyTrap();
IConsume consumeInterface;
consumeInterface = myVenusFlyTrap;
consumeInterface.EatFood();
```

In the preceding code snippets, calling `consumeInterface.EatFood()` results in the `EatFood()` method of the `Cow`, `Chicken`, or `VenusFlyTrap` class being called, depending on which instance has been assigned to the interface type variable.

Note here that derived classes inherit the interfaces supported by their base classes. In the first of the preceding examples, it may be that either `Animal` supports `IConsume` or that both `Cow` and `Chicken` support `IConsume`. Remember that classes with a base class in common do not necessarily have interfaces in common, and vice versa.

Relationships Between Objects

Inheritance is a simple relationship between objects that results in a base class being completely exposed by a derived class, where the derived class may also have some access to the inner workings of its base class (through protected members). There are other situations in which relationships between objects become important.

This section takes a brief look at the following:

- **Containment:** One class contains another. This is similar to inheritance but allows the containing class to control access to members of the contained class and even perform additional processing before using members of a contained class.
- **Collections:** One class acts as a container for multiple instances of another class. This is similar to having arrays of objects, but collections have additional functionality, including indexing, sorting, resizing, and more.

Containment

Containment is simple to achieve by using a member field to hold an object instance. This member field might be public, in which case users of the container object have access to its exposed methods and properties, much like with inheritance. However, you won't have access to the internals of the class via the derived class, as you would with inheritance.

Alternatively, you can make the contained member object a private member. If you do this, then none of its members will be accessible directly by users, even if they are public. Instead, you can provide access to these members using members of the containing class. This means that you have complete control over which members of the contained class to expose, if any, and you can perform additional processing in the containing class members before accessing the contained class members.

For example, a `Cow` class might contain an `Udder` class with the public method `Milk()`. The `Cow` object could call this method as required, perhaps as part of its `SupplyMilk()` method, but these details will not be apparent (or important) to users of the `Cow` object.

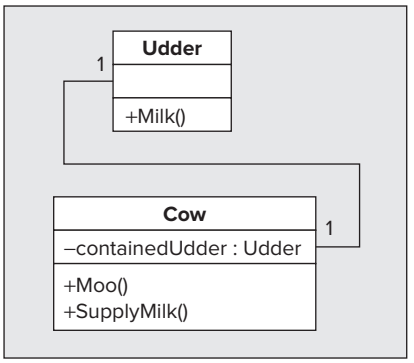

FIGURE 8-10

Contained classes may be visualized in UML using an association line. For simple containment, you label the ends of the lines with 1s, showing a one-to-one relationship (one `Cow` instance will contain one `Udder` instance). You can also show the contained `Udder` class instance as a private field of the `Cow` class for clarity (see Figure 8-10).

Collections

Chapter 5 described how you can use arrays to store multiple variables of the same type. This also works for objects (remember, the variable types you have been using are really objects, so this is no real surprise). Here's an example:

```
Animal[] animals = new Animal[5];
```

A collection is basically an array with bells and whistles. Collections are implemented as classes in much the same way as other objects. They are often named in the plural form of the objects they store — for example, a class called `Animals` might contain a collection of `Animal` objects.

The main difference from arrays is that collections usually implement additional functionality, such as `Add()` and `Remove()` methods to add and remove items to and from the collection. There is also usually an `Item` property that returns an object based on its index. More often than not this property is implemented in such a way as to allow more sophisticated access. For example, it would be possible to design `Animals` so that a given `Animal` object could be accessed by its name.

In UML you can visualize this as shown in Figure 8-11.

Members are not included in Figure 8-11 because it's the relationship that is being illustrated. The numbers on the ends of the connecting lines show that one `Animals` object will contain zero or more `Animal` objects. You'll take a more detailed look at collections in Chapter 11.

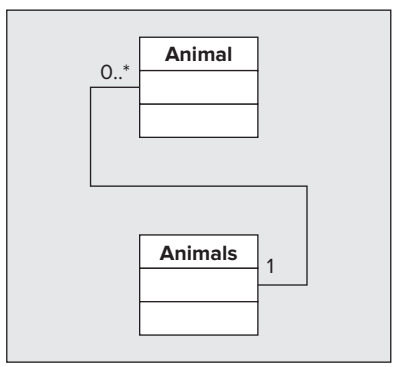

FIGURE 8-11

Operator Overloading

Earlier in the book, you saw how operators can be used to manipulate simple variable types. There are times when it is logical to use operators with objects instantiated from your own classes. This is possible because classes can contain instructions regarding how operators should be treated.

For example, you might add a new property to the `Animal` class called `Weight`. You could then compare animal weights using the following:

```
if (cowA.Weight > cowB.Weight)
{
   ...
}
```

Using operator overloading, you can provide logic that uses the `Weight` property implicitly in your code, so that you can write code such as the following:

```
if (cowA > cowB)
{
   ...
}
```

Here, the greater-than operator (`>`) has been *overloaded*. An overloaded operator is one for which you have written the code to perform the operation involved — this code is added to the class definition of one of the classes that it operates on. In the preceding example, you are using two `Cow` objects, so the operator overload definition is contained in the `Cow` class. You can also overload operators to work with different classes in the same way, where one (or both) of the class definitions contains the code to achieve this.

You can only overload existing C# operators in this way; you can't create new ones. However, you can provide implementations for both unary and binary usages of operators such as `+`. You'll see how to do this in C# in Chapter 13.

Events

Objects may raise (and consume) *events* as part of their processing. Events are important occurrences that you can act on in other parts of code, similar to (but more powerful than) exceptions. You might, for example, want some specific code to execute when an `Animal` object is added to an `Animals` collection, where that code isn't part of either the `Animals` class or the code that calls the `Add()` method. To do this, you need to add an *event handler* to your code, which is a special kind of function that is called when the event occurs. You also need to configure this handler to listen for the event you are interested in.

You can create *event-driven applications,* which are far more prolific than you might think. For example, bear in mind that Windows-based applications are entirely dependent on events. Every button click or scroll bar drag you perform is achieved through event handling, as the events are triggered by the mouse or keyboard.

Later in this chapter you will see how this works in Windows applications, and there is a more in-depth discussion of events in Chapter 13.