.NET Book Zero

What the C or C++ Programmer Needs to Know about C# and the .NET Framework

by

Charles Petzold

www.charlespetzold.com

Version 1.1, January 1, 2007

Table of Contents

Chapter 1. Why This Book?

Some books have a Chapter Zero. That's the chapter with the stuff the reader needs to know before reading Chapter One. Chapter Zero might be a refresher course in subjects the reader once knew but has now forgotten, or it might be a quick-and-dirty summary of prerequisites for the rest of the book.

This book originated as a Chapter Zero in my book *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation* (Microsoft Press, 2006), which is about the new Windows client programming platform that's part of Microsoft .NET 3.0 and Microsoft Windows Vista.

I wanted *Applications = Code + Markup* to focus almost exclusively on the Windows Presentation Foundation. I knew there was enough to cover without going into the basics of general .NET programming and C#. Yet, I wasn't sure how much .NET my readers would know. I started writing a Chapter Zero for the book that would summarize all the basics of .NET and C# for the C and C++ programmers who might be coming to .NET for the very first time.

It soon became evident that this Chapter Zero would be very long. It occurred to me that I could extract the material and make it a book on its own. And so I did and this is it. What you have in your hands (or are reading on a screen) is an introduction to C# and those topics in .NET that are typically found in all .NET programming.

C# is a modern type-safe and object-oriented programming language based on the syntax of C and (to a certain extent) C++ and Java. Even if you're an experienced C++ programmer, you might be in for a few surprises. You may think you know the difference between a *class* and a *struct*, for example, but the difference between a *class* and a *struct* in C# is completely different from C++. (That difference is actually one of the lamest features of C++ and one of the most profound features of C#.) For that reason and others, I approach object-oriented programming concepts in this book almost as if you're learning about them for the very first time.

However, I do expect you to have some programming experience with a C-family language. If you're learning C# as a first programming language, you might be better off with a slower, gentler introduction, such as my book *Programming in the Key of C#: A Primer for Aspiring Programmers* (Microsoft Press, 2003).

If you'd like to reference this book on your own Web site, I ask that you do so using a link to the page on my Web site where this book is found. That page is:

http://www.charlespetzold.com/dotnet

That's the page where people can find the latest version of the book and download the source code I show in the pages that follow.

If you like this book, perhaps you'd like to read some other books I've written. Come to my web site, www.charlespetzold.com and check them out. These other books aren't free, alas. They will cost you money. But you will be pleased to know that I receive a small percentage of the price you pay for each book. That money helps me pay my rent and feed myself, and enables me to write more books in the future.

In writing this book, I've drawn upon some of my earlier writing about C# and the .NET Framework. Some of the earlier chapters are revisions of Chapter 1 in *Programming Microsoft Windows with C#* (Microsoft Press, 2001), which is an introduction to Windows Forms programming. Some of the later chapters were drawn from appendices of that book. The chapters specific to the object-oriented programming aspects of C# were drawn from my book *Programming in the Key of C#*.

As a .NET programmer, you'll probably specialize in a particular aspect of .NET, and you'll likely buy a couple books on that subject. But there are two essential books that every C# and .NET programmer should have.

The first essential book is *The C# Programming Language* by Anders Hejlsberg, Scott Wiltamuth, and Peter Golde (2nd edition, Addison-Wesley, 2006). This book is the official technical specification of the C# language. It is certainly not a tutorial for learning the language, but a great book to read *after* you've become adept at C# programming.

Downloadable versions of *The C# Programming Language* are available under the title *C# Language Specification* from this Web page:

http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx

Because the online title *C# Language Specification* is actually more accurate than the book title *The C# Programming Language*, I will refer to the online title rather than the book title when I sometimes refer to the book using chapter and section numbers.

The second essential .NET book is Jeffrey Richter's *CLR via C#* (Microsoft Press, 2006), which is actually the second edition of *Applied Microsoft .NET Framework Programming*. There are many subtle and interesting aspects of .NET programming that Richter's book explores in much more depth than you'll find in the pages ahead that I've written.

In *.NET Book Zero* and my other books, I tend to focus more on the C# language and the .NET Framework class libraries rather than Microsoft Visual Studio. As you probably know, Visual Studio is the primary programming environment for creating .NET applications. You might want to supplement your C# and .NET studies with a book specific to Visual Studio.

Because this book is intended to teach C# and the rudiments of .NET, much of the code I show in the pages ahead targets the traditional (and largely obsolete) command line using character-mode programming interfaces. I am well aware that you'll probably eventually be coding for graphical environments, and that you might consider learning about character-mode programming to be a complete waste of your time. This is not so. The character-formatting techniques you learn here are directly applicable to graphical programming as well.

This book is written in tutorial style, which means that it is intended to be read sequentially. The reader is encouraged to type in the programs as they are encountered in the book, to run them, and experiment with them.

* * *

Version 1.0 of this book was posted to *www.charlespetzold.com/dotnet* on December 4, 2006.

Version 1.1 was posted on January 1, 2007. It incorporated many minor corrections reported by Larry Danielle, Paul Dougherty, Paul Duggan, David Evans, Thorsten Franz, Konstantin Korobkov, Tyson Maxwell, Ryan McFarren, and Larry Smith.

Chapter 2. Why .NET?

The Microsoft .NET Framework (which I'll often refer to with the simpler term .NET) is a collection of software technologies that began emerging from Microsoft Corporation around the turn of the century. The first version of .NET was released early in 2002, and version 1.1 came out in 2003. Version 2.0 was released late in 2005, and Version 3.0 followed in late 2006. A good overview of the .NET releases can be found in the Wikipedia entry on the Microsoft .NET Framework:

http://en.wikipedia.org/wiki/.NET_Framework

From the end-user's perspective, .NET is fairly invisible. The savvier user might feel enlightened to know that .NET is basically a collection of dynamic link libraries. These DLLs might already be installed along with Windows XP on a new machine, or they might be installed during the process of installing an application that uses .NET. The latest version of Windows—Microsoft Windows Vista—includes the .NET Framework 3.0 as an intrinsic part of its architecture.

From the programmer's perspective, .NET is a huge class library that contains everything you need to write Web applications or client applications—the type of programs sometimes called "regular old Windows apps."

If you are a programmer, and you write (or want to write) Web applications or Windows client applications, and you haven't yet started exploring .NET, then reading this book is a good move. Congratulations on getting started!

You can program for .NET in a variety of programming languages. However, any language you use for .NET programming must meet a set of minimum requirements to order to use the .NET class libraries. These requirements are known as the .NET Common Language Specification or CLS. Related to the CLS is the .NET Common Type System (CTS) which defines the basic data types (such as integer, floating point, and string) that .NET languages support. The CLS and CTS are in turn part of the Common Language Infrastructure (CLI). The CLI is an ISO standard and an ECMA standard.

When you compile one of your .NET programs, the program is generally compiled to a processor-independent intermediate language that resembles machine code. This intermediate language was once called Microsoft Intermediate Language (MSIL), and it's still often known by that name. Sometimes it's just called IL. But the most proper term is now the Common Intermediate Language (CIL).

When a .NET program is run on a particular machine, the CIL is compiled to the native code of the processor by a just-in-time (JIT) compiler. This two-stage compilation potentially allows for portability among various platforms and processors.

The just-in-time compilation is performed by the .NET Common Language Runtime (CLR), which is part of the .NET system installed on end-user's machines. The CLR manages the execution of .NET programs, and can prevent programs from causing damage to the user's machine. Thus, when you are programming for .NET you are said to be writing "managed code."

One important aspect of managed code involves the management of memory. As object-oriented programming and class libraries have become more complex over recent years, common problems have arisen involving memory allocation. Very often it's not clear who is responsible for freeing a particular memory block. For that reason, the CLR implements garbage collection. The CLR can determine if a particular block of memory can no longer be referenced by a program, and then free such blocks of memory if required.

Microsoft makes available several languages to the .NET programmer. Which one you use is mostly a matter of personal taste. Some people program for .NET using Visual Basic .NET. Others use Managed C++, more formally known now as C++/CLI.

However, most .NET programmers have come to know and love C#, the programming language designed in conjunction with .NET largely under the guidance of Anders Hejlsberg. That's the language I'll be describing in the pages that follow.

C# incorporates much of the basic expression and statement syntax of C, and has a rather cleaner object-oriented programming syntax than C++. The big difference that veteran programmers will discover is that C# does not require you to mess around with pointers. Traditional C-like pointers are supported in C# syntax, but they are normally relegated to interoperability with existing code. (I won't be discussing C# pointers in this book; if you want that information, you can find it elsewhere.)

Rather than pointers, the .NET and C# programmer works with "references," and these references are usually implied rather than being syntactically explicit. It is part of becoming a good C# programmer that you learn when you are working with a reference and when you are not.

It is never too early to start learning the C# and .NET mantra:

Classes are reference types; structures are value types.

Chapter 3. Runtimes and SDKs

To run .NET programs on your machine, you'll need to have some software installed that is variously known as the .NET "runtime" or "runtime components" or "redistributable" or "redistributable package." The term "redistributable" means that a software developer like yourself can distribute the .NET runtime if it's part of an installation for an application that requires the .NET Framework to run.

You'll need the .NET runtime components to run .NET programs. To develop .NET programs on your machine, you'll also need to install the .NET Framework Software Development Kit (SDK). Both the runtime and the SDK are free and both are generally downloadable from the same or related Web pages.

To determine what versions of .NET (if any) are currently installed on your machine, the following Knowledge Base article can help:

http://support.microsoft.com/kb/318785

For installations of the .NET Framework 1.1 and the SDK, go to this page:

http://msdn2.microsoft.com/netframework/aa569264.aspx

Although this page includes a redistributable for .NET 1.1, it is recommended that end users install the .NET 1.1 runtime components as part of a Windows update.

For the .NET Framework 2.0, go here:

http://msdn2.microsoft.com/netframework/aa731542.aspx

For the .NET Framework 3.0, go here:

http://msdn2.microsoft.com/windowsvista/aa904955.aspx

The SDK is referred to on this page as the "Windows SDK." As of this writing, .NET version 3.0 is fairly recent, but it is likely to become the "standard" version of .NET because it is built into Microsoft Windows Vista. However, you may want to target a lesser version of .NET if you know that it's supported by an existing user base.

The most recent version of Microsoft Visual Studio is Visual Studio 2005, which incorporates the .NET Framework 2.0 SDK. The next version of Visual Studio will incorporate the .NET Framework 3.0 SDK. Meanwhile, if you want to do .NET 3.0 programming with Visual Studio, you'll need to install the 3.0 SDK along with Visual Studio 2005. If you need to program for a specific subsystem of .NET 3.0 (such as the Windows Presen-

tation Foundation or the Windows Communication Foundation or the Windows Workflow Foundation) you can install extensions to Visual Studio 2005. These are available as links from the .NET Framework 3.0 page.

Microsoft also makes available a free Visual C# 2005 Express Edition that you can download here:

http://msdn.microsoft.com/vstudio/express/visualcsharp

This package installs the .NET 2.0 runtime and a good chunk of the SDK. (You can install the 2.0 SDK in addition to the Visual C# Express Edition.) The installation asks if you want to install MSDN, which stands for Microsoft Developer Network and refers to documentation that includes the .NET class libraries. You'll very likely want to install this documentation.

Strictly speaking, you don't need either Visual Studio or Visual C# to program for .NET. The .NET Framework SDK comes with a command-line version of the C# compiler, and you can use that. However, Visual Studio and Visual C# simplify several aspects of .NET programming.

Besides the compiler itself, perhaps the most important part of the SDK is the documentation of the .NET class libraries. When you install one of the SDKs, the SDK itself appears on the Windows start menu, and a Documentation item appears within that group. (If you've only installed Visual C# 2005 Express Edition, you can bring up the documentation by selecting Contents from the Help menu of Visual C#.)

The .NET documentation is displayed by the Document Explorer application. On the left side of the Document Explorer window is a pane that you can switch between Content and Index views with a tab at the bottom. The pane on the right side shows information on the selected item.

Select the Content tab. I want you to find the documentation of the .NET class libraries. If you've installed the .NET 1.1 SDK, you're looking for the Class Library heading in the following hierarchy:

.NET Framework SDK
    Reference
        Class Library

With a later SDK, the hierarchy is a bit shorter:

.NET Framework SDK
    Class Library

Or:

.NET Framework Development
Class Library

When you find it, you'll know it by the large list of entries. Many of the early entries begin with the word *Microsoft*. The later entries begin with the word *System*. What you're seeing here is the basic class documentation of the .NET Framework, and you'll be spending lots of time with it. You can also access the .NET Framework documentation online at this page:

http://msdn2.microsoft.com/library/aa388745.aspx

The top-level entries in this long list that begin with the words *Microsoft* or *System* are known as *namespaces*. The namespaces serve to separate the .NET Framework into functional groups. For example, *System.Windows.Forms* is the basic namespace for Windows Forms. Namespaces also help avoid problems resulting from duplicate class names. The .NET Framework can have different classes with the same names. If these classes are in different namespaces, there's no name clash. There are three classes named *Timer*, for example, all in different namespaces.

Some of these namespaces will become an intimate part of your life; others you'll probably never get to know. As the popular tee-shirt says, "So many .NET namespaces… so little time."

The most important namespace is *System*, and that's the namespace I'll be referring to most in this book. A few other namespaces are often useful, even in traditional character-mode programs. The *System.Globalization* namespace contains classes that help you tailor your programs to an international market. The *System.Collections* and *System.Collections.Generic* contain classes that help you store information in familiar collections such as queues, stacks, and dictionaries. The *System.IO* namespace contains essential classes for working with files and streams, and *System.Xml* supplements those classes for working with XML.

If you open one of these namespaces in the documentation, you'll see a number of *types* defined in the namespace. Most of these types are classes. Some are structures. Others are interfaces, enumerations, and delegates. You'll learn more about these five types in the pages ahead.

Open up a class or structure, and you'll see *members* of that type. These members can include constructors, fields, methods, properties, and events, which you'll also learn more about in the pages ahead.

Whenever you're doing .NET programming (or whenever you're reading this book or any other .NET programming book) you'll probably want to have the .NET documentation open and ready for browsing.

To quickly find a particular item in the class documentation, click the Index tab in the left pane. In the Look For field, enter what you're looking for: "Timer class," for example. Select "about Timer class" in the list. Over at the right on the bottom, you'll see the three *Timer* classes with their namespaces in parentheses. Select the one you want, and the first page of the class documentation will appear. You can then click the Sync With Table Of Contents button on the toolbar to get back to the Contents view and continue exploring the particular class. (In the .NET Framework 1.1 SDK, it works a little differently. There is no separate pane for index results; the three *Timer* classes are listed separately in the index.)

Besides providing all the class documentation of the .NET Framework, another important role of the .NET Framework documentation is the teaching of humility. You will never, ever, come close to any type of familiarity with the entire .NET class library. (But you can always try.)

Chapter 4. Edit, Compile, Run, Disassemble

A file containing C# code generally has the filename extension .cs for "C Sharp." Here's a simple example (the boldfaced filename at the top is not part of the program):

FirstProgram.cs

```
//-----------------------------------------
// FirstProgram.cs (c) 2006 by Charles Petzold
//-----------------------------------------

class FirstProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, Microsoft .NET Framework!");
    }
}
```

Let's first try to create, compile, and run this program, and then I'll discuss its structure and contents.

Although you'll probably eventually use Microsoft Visual Studio to develop .NET programs, that's not your only option. You can actually edit, compile, and run .NET programs from the MS-DOS command line. In many respects, compiling a C# program on the command line is quite similar to the way you might have compiled a C program on the command line two decades ago.

Compiling .NET programs on the MS-DOS command line might seem like an odd and eccentric practice in modern graphical environments like Windows, but I think it's important for the beginning .NET programmer to try it just once. At the very least, you'll be disabused of the notion that you need the powerful resources of Visual Studio to compile every .NET program you'll ever write.

(Some information in this chapter does not apply to the .NET 1.1 SDK. If that's what you're using, you'll want to select the Tools item in the Microsoft .NET Framework SDK v1.1 entry in the Windows start menu for information about the command line, the IL disassembler, and the IL assembler.)

Both Visual Studio 2005 and the .NET 2.0 and 3.0 SDKs create entries in the Windows start menu for running command-line windows. This is what you should use. It's harder to use the regular Windows Command Prompt window for compilations because it doesn't have the proper environment variables set so that MS-DOS can locate the C# compiler.

If you run one of these command-line windows, you can then navigate to a particular directory where you want to store your programs. On the command line, type

```
notepad
```

and Windows Notepad will run. Or, you can type a filename as an argument to Windows Notepad like this:

```
notepad firstprogram.cs
```

Then Notepad will ask you if you want to create that file.

In Notepad, type in the program shown above. C# is a case-sensitive language. Make sure that you type the words *class*, *public*, *static*, and *void* entirely in lowercase. Make sure you type the words *Main*, *System*, and *Console*, with an initial capital but the rest in lower-case. Make sure that *WriteLine* has an initial capital and an embedded capital. You can type *FirstProgram* whatever way you want (or you can use a different name), but don't embed a blank in the name and don't begin the name with a number. You don't need to include the lines that begin with double slashes.

Save the file from Notepad with the name firstprogram.cs, or something else if you prefer. (You don't need to exit Notepad at this point, but you do need to save the file.) Then, on the command-line, run the C# compiler, which is a program named csc.exe:

```
csc firstprogram.cs
```

The C# compiler reads your source code and (if all is well) emits a file named firstprogram.exe, which you can run like this:

```
firstprogram
```

The program displays a text greeting and then terminates.

I mentioned in the second chapter that a .NET executable actually contains Common Intermediate Language (CIL) code. The .NET SDK includes a tool called the IL Disassembler (ildasm.exe) that disassembles a .NET executable and shows you the CIL statements. From the Windows start menu, find the SDK group, and then a tool named IL Disassembler. Run it. Or, just enter

```
ildasm
```

on the command line. From the File Open dialog box, navigate to the directory you've been using, and load FirstProgram.exe. Open the *FirstProgram* class and double-click *Main*. That's your program in CIL. The *ldstr* command loads a text string on the stack, and then a *call* command calls *System.Console.WriteLine* (but with a syntax more reminiscent of C++) to display the string. When you run the program, the .NET Common Language Runtime (CLR) compiles the CIL into machine code appropriate for your particular processor.

If learning CIL appeals to you, you can discover more about it here:

http://www.ecma-international.org/publications/standards/Ecma-335.htm

The .NET SDK includes an assembler program for CIL named ilasm.exe. Programs written directly in CIL are just as managed and just as portable as programs written in C#.

Of course, most .NET programmers don't know any CIL at all, and even fewer know enough CIL to be able to actually code in it. However, it is sometimes instructive and revealing to examine the CIL that the C# compiler emits, and in this book I'll occasionally call your attention to it.

Now let's jump from command-line programming to the opposite extreme by running Visual Studio 2005 or Visual C# 2005 Express Edition.

From the menu select File, then New and Project. In Visual Studio, first select Visual C# and Windows at the left. In either edition, select Empty Project on the right. Give the project a name (FirstProgram, let's say). In Visual Studio, you'll need to select a directory location for the project and uncheck the Create Directory For Solution checkbox. In Visual C# Express Edition, you select the directory when you save the project.

In the Solution Explorer on the right, right-click the FirstProgram project and select Add and New Item. (Or, select Add New Item from the Project menu.) Select Code File and give the file a name of FirstProgram.cs.

Now type in the program shown above. As you type, you'll see that Visual Studio trys to anticipate what you need. When you type *System* and a period, for example, it will give you a list of types in that namespace, and when you type *Console* and a period, you'll get a list of members of the *Console* class. This is Visual Studio's Intellisense, and you might come to find it addictive, and then hate yourself from relying on it so much.

You can compile and run the program by selecting Start Without Debugging from the Debug menu or by pressing Ctrl-F5. The program will compile and run in a command-line window.

What you've done here is to create a Visual Studio *project* named FirstProgram, which occupies the FirstProgram directory. A project generally creates a single executable file or a single dynamic link library. (In Visual Studio, multiple related projects can also be bundled into *solutions*.) A project can contain one or more C# source code files. In the simplest case, a project contains one C# file, and for convenience the C# file is generally given the same name as the project but with a .cs extension.

Back on the command line or in Windows Explorer, you can see that Visual Studio has created a project file in the FirstProgram directory named FirstProgram.csproj. This is an XML file that references the .cs

file and contains all the other information Visual Studio needs to maintain the project and compile it.

During compilation, Visual Studio has also created some intermediate files in a subdirectory of FirstProgram named *obj*. The executable file is found in *bin* in a subdirectory named either *Release* or *Debug* depending on the configuration you've chosen in Visual Studio.

If you're running .NET 3.0, go back to the command line. Make sure the FirstProgram.csproj file is in the current directory and run:

```
msbuild firstprogram.csproj
```

The MSBuild program will compile the project and (by default) deposit the executable file in the *bin\Debug* directory.

The MSBuild program became necessary in .NET 3.0 partially because Windows Presentation Foundation programs can be built from both C# files and XAML (Extensible Application Markup Language) files. The MSBuild program invokes the C# compiler and other tools to assemble an entire executable. You can write your own .csproj project files, if you want.

Between the extremes of the command prompt and Visual Studio are other .NET programming tools, such as my own KeyOfCSharp.exe, which you can download here:

http://www.charlespetzold.com/keycs

If you want to run the sample programs shown in this book without typing them in, you can download all the source code from the same page where you found this book:

http://www.charlespetzold.com/dotnet

However, you'll better accustom your brain and fingers to C# code by typing in the code yourself.

Let's look at the program listing again:

FirstProgram.cs

```
//-----------------------------------------
// FirstProgram.cs (c) 2006 by Charles Petzold
//-----------------------------------------

class FirstProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, Microsoft .NET Framework!");
    }
}
```

At the top are a few single-line comments beginning with the familiar double slashes. C# also supports multi-line or partial-line comments delimited by /* and */.

All code in a C# program must be in either a class or a structure. This particular program defines a class (denoted by the keyword *class*) named *FirstProgram*:

```
class FirstProgram
{
    // contents of the class
}
```

Curly brackets delimit the contents of the class. You can change that class to a structure using the keyword *struct*:

```
struct FirstProgram
{
   // contents of the structure
}
```

The program will compile and run the same.

It is common to define the class or structure with the *public* keyword:

```
public class FirstProgram
{
    // contents of the class
}
```

However, using the *public* keyword with a class is not generally required in program code. (There are some cases where it is required.) The *public* keyword applied to a class is generally found much more in code that contributes to dynamic link libraries.

When creating this project in Visual Studio, I've used a project name that is the same as the C# file name, which is the same as the name of the class defined in that file. None of these name matches is required. In fact, a C# file can contain multiple class definitions, none of which match the file name. A class can also be split between multiple files, none of whose names need match the class name. None of these names need to be the same as the project name.

Just to avoid confusion, I generally like to restrict my C# source code files to just a single class and structure, and to use a file name that matches the class or structure name. (But I sometimes break this rule in this book.)

In the *FirstProgram* class (or structure) is a single method named *Main*. The entry point to a C# program is always a method named *Main*, and it must have an initial capital. C# is a case-sensitive language.

The *Main* method is defined with the *public* and *static* keywords:

```
public static void Main()
{
    // contents of the method
}
```

The *public* keyword indicates that the method is visible from outside the class in which it is defined. The *public* keyword is not actually required for the *Main* method, and the program will compile and run fine without it. Sometimes I use *public* with *Main* and sometimes not. It's a mystery.

The *static* keyword means that this method is associated with the class itself rather than an instance of that class. A class is basically an ice cream dispenser, and instances of the class are sundaes. Unfortunately, this simple program isn't making any sundaes. There is no *instance* keyword, however, because *static* methods are generally the exception rather than the rule. The world has many more sundaes than ice cream dispensers, and generally sundaes are more interesting.

This particular *Main* method has no parameters — indicated by the empty parentheses following *Main*—and doesn't return anything to the caller, indicated by the keyword *void*. (You can also define *Main* to have a parameter that is an array of text strings, which are set to the command-line arguments of the program. *Main* can also return an integer as a termination code. See the *C# Language Specification*, §3.1 for details.)

The body of a method is delimited by curly brackets. The entire body of this *Main* method is the statement:

```
System.Console.WriteLine("Hello, Microsoft .NET Framework!");
```

As in C and C++, statements in C# are terminated by semicolons. This statement is a method call. The argument is a string literal enclosed in double-quotation marks. String literals in C# are restricted to a single line. In other words, the two quotation marks delimiting the string must appear on the same line. (If you need to break up a long string on multiple lines, you can concatenate multiple string literals using the plus operator, as I'll demonstrate in the next chapter.)

Although string literals must appear on one line, C# can otherwise be freely formatted. This is allowed:

```
class
    FirstProgram
                {
public
             static
     void
                Main
        (
                        )
    {
  System
                    .
```

```
            Console
          .
              WriteLine
                       (
    "Hello, Microsoft .NET Framework!"
                   )
                    ;
          }
                   }
```

So is this:

```
class FirstProgram{public static void Main(
){System.Console.WriteLine("Hello, Microsoft .NET Framework!");}}
```

If you code like this, however, nobody will be your friend.

FirstProgram doesn't do much except make a call to a method named *System.Console.WriteLine*. That's a fully-qualified method name. Like romance novelists, methods in the .NET Framework generally have three names:

- *System* is a namespace.
- *Console* is a class in that namespace.
- *WriteLine* is a method in that class.

In the .NET class documentation you'll find that the *Console* class actually has many methods named *WriteLine*. These various versions of the *WriteLine* method are known as *overloads*. The one I'm using in this particular program is defined like so (eliminating line breaks provided in the documentation):

```
public static void WriteLine(string value)
```

There's that keyword *static* again, and what it means here is that *WriteLine* is a method associated with the *Console* class rather than an instance of the *Console* class. The *static* keyword means the method must be referenced by prefacing it with the name of the class in which it's defined, separated by a period. The class is prefaced with the namespace in which the class is defined, also separated with a period.

Where is the code for *System.Console.WriteLine*. which is the code that actually puts the text on the console? If you look at the first page of the documentation for the *Console* class, you'll see near the top the following:

Assembly: mscorlib (in mscorlib.dll)

This indicates that the code for the *Console* class is located in an assembly named mscorlib. An assembly can consist of multiple files, but in this case it's only one file, which is the dynamic link library mscorlib.dll. The mscorlib.dll file is very important in .NET. The file name at one time stood for "Microsoft Common Object Runtime Library" but now it stands for "Multilanguage Standard Common Object Runtime Library." This is

the main DLL for class libraries in .NET, and it contains all the basic .NET classes and structures.

As you know, when you compile a C or C++ program, you generally need an #*include* directive at the top that references a header file. The include file provides function prototypes to the compiler.

The C# compiler does not need header files. During compilation, the C# compiler access the mscorlib.dll file directly and obtains information from metadata in that file concerning all the classes and other types defined therein. The C# compiler is able to establish that mscorlib.dll does indeed contain a class named *Console* in a namespace named *System* with a method named *WriteLine* that accepts a single argument of type *string*. The C# compiler can determine that the *WriteLine* call is valid, and the compiler establishes a reference to the mscorlib assembly in the executable.

Intellisense also works by referencing mscorlib.dll and getting information from the DLL about the namespaces, types, and members.

As you probably know, compiling a C or C++ program is just the first step in creating an executable. You must then (either explicitly or implicitly) run a linker that accesses library files. Traditionally, code in the standard runtime libraries is inserted right into the executable. For code in DLL's, only references are inserted.

The C# compiler doesn't require library files. Because the compiler is already accessing the actual DLL, it can insert references to that DLL into the executable. At the time the program is run, the CLR links the program with the actual method call in mscorlib.dll.

Many of the basic classes and structures are included in mscorlib.dll. As you go beyond the command line, you'll start encountering classes that are stored in other DLLs. For example, classes in the *System.Windows.Forms* namespace are stored in the assembly system.windows.forms, which is the DLL system.windows.forms.dll.

The C# compiler will access mscorlib.dll by default, but for other DLLs, you'll need to tell the compiler the assembly in which the classes are located. These are known as *references*. In Visual Studio, right click References under the project name in the Solution Explorer, and select Add Reference. Or, select Add Reference from the Project menu. (For the command line compiler, you specify references with the /r compiler switch.)

It may seem like a big bother to type *System.Console.WriteLine* just to display a line of text, and that's why the C# language supports a directive that reduces your typing a bit. This program is functionally equivalent to the program shown earlier:

SecondProgram.cs

```
//-----------------------------------------
// SecondProgram.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class SecondProgram
{
    public static void Main()
    {
        Console.WriteLine("Hello, Microsoft .NET Framework!");
    }
}
```

The *Console.WriteLine* call is no longer prceded with the *System* namespace. This is made possible by the line near the top that begins with the keyword *using*. This line is called a *directive* because it's not, strictly speaking, a statement. It must appear before any type definition in the file, such as a class. The *using* directive basically tells the C# compiler that if it can't find a static method named *Console.WriteLine*, it should try appending *System* to the front to make *System.Console.WriteLine* and try looking for that.

Of course, the *using* directive hasn't exactly reduced the size of the program, but if you had very many *WriteLine* calls, it certainly would. All the sample programs from now on will have a *using* directive for the *System* namespace and occasionally other namespaces as well.

The *using* directive is not like a header file, and it's not like a library file. It doesn't reference a file. The *using* directive only indicates a namespace, and having *using* directives is never required in a C# program.

A slightly different form of the *using* directive defines an alias that lets you decrease your repetitive typing even more.

ThirdProgram.cs

```
//-----------------------------------------
// ThirdProgram.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using C = System.Console;

class ThirdProgram
{
    public static void Main()
    {
        C.WriteLine("Hello, Microsoft .NET Framework!");
    }
}
```

Now any reference to the *System.Console* class can be shortened to just a capital *C*. That's about as succinct as you're going to get here. (The next step would involve defining your own short-named method that then calls *WriteLine*.)

I need to warn you how limited the alias facility is: The *using* directive does *not* perform a substitution of *System.Console* for any and all occurrences of a capital C in your code. In the *using* directive, the right side of the equals sign must be a namespace or type, and this particular example only comes into play if the compiler cannot find a class named *C* with a method named *WriteLine*.

Also keep in mind that C# culture does not encourage the use of techniques like this to make your code look more obscure than it should be. The use of this form of the *using* statement is primarily for a situation where you need to reference classes with the same name from two different namespaces.

For example, suppose you purchase two helpful class libraries in the form of DLLs from Bovary Enterprises and Karenina Software. Both these libraries contain a class named *SuperString* that is implemented entirely differently in each DLL but is useful to you in both versions. Using both *SuperString* clases is not a problem because both companies defined unique namespace names for their classes.

The people at Bovary put their *SuperString* class in a namespace named *BovaryEnterprises.VeryUsefulLibrary*. Yes, the namespace contains an embedded period, and it's in accordance with accepted practices. The company name goes first, followed by a product name. The code developed at Bovary looked something like this:

```
namespace BoveryEnterprises.VeryUsefulLibrary
{
    public class SuperString
    {
        ...
    }
}
```

The clever programmers at Karenina also used the accepted naming convention and put their *SuperString* class in the namespace *KareninaSoftware.HandyDandyLibrary*.

So, when using both these DLLs in your own program, you can reference either *SuperString* class simply by using the fully-qualified name, which is either

```
BovaryEnterprises.VeryUsefulLibrary.SuperString
```

or:

```
KareninaSoftware.HandyDandyLibrary.SuperString
```

And here's where the alias form of the *using* directive comes into play. To simplify your typing, you can include the following two *using* directives in your program:

```
using Emma = BovaryEnterprises.VeryUsefulLibrary;
using Anna = KareninaSoftware.HandyDandyLibrary;
```

Now you can refer to the two classes as

```
Emma.SuperString
```

and:

```
Anna.SuperString
```

If you are writing code for a DLL, and particularly if you intend to make this DLL available to others, you should put everything in a namespace that identifies your company and product.

You can also use namespace definitions in your non-DLL program code, but here it's not so vital. For the first couple of .NET programming books I wrote, I didn't use namespaces at all in my programs. In my recent book on the Windows Presentation Foundation, I used namespaces in my program code that consisted of my name followed by the project name. I did this for two reasons. Most importantly, when integrating C# code with XAML, it helps for the program code to be in a namespace. Also, when one project references code from another project (as I do in my WPF book), the namespace helps identify where the referenced code comes from.

The following program shows how you can put your own program code inside a namespace definition.

FourthProgram.cs

```
//-----------------------------------------------
// FourthProgram.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

namespace Petzold.FourthProgram
{
    class FourthProgram
    {
        public static void Main()
        {
            Console.WriteLine("Hello, Microsoft .NET Framework!");
        }
    }
}
```

However, in this little book, that's the last you'll see of a namespace definition.

Chapter 5. Strings and the Console

In the previous chapter, the argument passed to the *Console.WriteLine* method looked like this:

```
"Hello, Microsoft .NET Framework"
```

That is known as a string literal. It consists of a bunch of characters delimited by double quotation marks. The characters are Unicode, which means that each character is represented by a 16-bit number. (More information about Unicode can be found at *www.unicode.org*.)

As in C and C++, the backslash character is interpreted as an escape character, and the character that follows is treated specially. This allows the embedding of characters in a character string that would otherwise not be possible. The following table shows the supported escape sequences with their Unicode equivalents in hexadecimal.

Escape Sequence	**Result**	**Unicode Encoding**
\0	Null	0x0000
\a	Alert (beep)	0x0007
\b	Backspace	0x0008
\t	Horizontal tab	0x0009
\n	New line	0x000A
\v	Vertical tab (printing)	0x000B
\f	Form feed (printing)	0x000C
\r	Carriage return	0x000D
\"	Double quote	0x0022
\'	Single quote	0x0027
\\	Backslash	0x005C
\uABCD \xABCD	Unicode character	0xABCD

I've never found it necessary to precede a single quote mark with a back-slash in a string. (You'll need to do so when defining a character literal because character literals are delimited by single quote marks.) The last entry in the table indicates how you can embed arbitrary Unicode characters in a character string. The ABCD characters stand for any 4-digit hexadecimal number. For example:

```
"Hello, Microsoft\x00AE .NET Framework"
```

Now the word "Microsoft" is followed by a ® symbol to make the lawyers happy. However, the console doesn't support non-ASCII characters very well, so if you actually make this change in the program from the last chapter, it will probably show up simply as a lower-case 'r'.

If you really, really, really want to see your program display an ® symbol, you can abandon the console and write a small Windows Forms program. Windows Forms is a Windows client platform supported under all versions of .NET.

TextWithUnicodeChar.cs

```
//-----------------------------------------------------
// TextWithUnicodeChar.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System.Windows.Forms;

class TextWithUnicodeChar
{
    public static void Main()
    {
        MessageBox.Show("Hello, Microsoft\x00AE .NET Framework");
    }
}
```

Show is a static method in the *MessageBox* class, which is in the *System.Windows.Forms* namespace. Without the *using* directive, you'd have to call this method with the horrific fully-qualified name:

```
System.Windows.Forms.MessageBox.Show(
            "Hello, Microsoft\x00AE .NET Framework");
```

The Windows Forms classes are in the System.Windows.Forms assembly, which is the System.Windows.Forms.dll file. To compile this program you need a reference to that assembly. In Visual Studio in the Solution Explorer, right click References and then Add Reference. (Or select Add Reference from the Project menu.) In the Add Reference dialog box, select the .NET tab and the System.Windows.Forms assembly. When compiling on the command line, use the /r switch to specify other assemblies.

The *MessageBox.Show* method displays a Windows message box with an OK button. When you click the OK button, the message box disappears from the screen, *MessageBox.Show* returns, and the program terminates.

Although the Windows Forms program correctly displays the ® symbol, keep in mind that not every font supports every Unicode character.

You can also use Unicode escape sequences in variable names. See the *C# Language Specification*, §2.4.1 for details.

In some cases you might want to encode a string literal with several backslashes. This is common with directory paths:

```
"\\Documents and Settings\\Charles\\Favorites"
```

You can alternatively use a type of string literal known as the *verbatim* string literal. You preface the first double quote with an @ sign:

```
@"\Documents and Settings\Charles\Favorites"
```

The backslash ceases to be an escape character so you only need one backslash for each separator. None of the normal escape sequences are allowed. If you need to embed a double quote in the string, use two double quotes in a row.

```
@"The symbol \ is called a ""backslash"""
```

Verbatim strings can begin on one line and continue to the next, although the resultant string will have embedded carriage return and line feed characters.

Unlike C and C++, C# supports a *string* data type for storing strings. Within a method such as *Main* you can declare a variable of type *string* using a declaration statement:

```
string str;
```

All variables must be declared before they are used. Variable names generally begin with letters or an underscore, and can also contain numbers, but the rules for what Unicode characters are allowed in a variable name are quite complex. (See the *C# Language Specification*, §2.4.2.) Certainly the variable name doesn't have to begin with the letters *str*, but I like to do that because it reminds me that this is a *string* variable.

You can optionally initialize that string when you declare it:

```
string str = "This is an initialized string";
```

Or you can set the value of the string after it's declared with an assignment statement:

```
string str;
str = "This is an assigned string";
```

There's no difference between initializing a string in a declaration statement and assigning it immediately after the declaration statement.

You can declare multiple string variables in a single declaration statement by separating them with commas:

```
string str1, str2, str3;
```

You can initialize all or some of these variables:

```
string str1, str2 = "initialized", str3;
```

Until a string variable is assigned a value, it is considered to be uninitialized, and the C# compiler will not allow that variable to be used. Here's an illegal sequence of statements:

```
string str;
```

```
Console.WriteLine(str);
```

The C# will complain about the "Use of unassigned local variable 'str'."

You can set a *string* variable to an empty string:

```
string str = "";
```

Or you can set the *string* variable to the C# keyword *null*:

```
string str = null;
```

In either case, the variable is now considered to be initialized, but in two distinctly different ways. In the first case, the *str* variable refers to a string that happens to have no characters. In the second case, the *str* variable is considered to have a *null* reference, which means that it doesn't refer to anything. In either case, *Console.WriteLine* will just display nothing for that string.

Here's a complete program that uses an initialized string in *Main*:

```
class Program
{
    static void Main()
    {
        string strDisplay = "Hello, Microsoft .NET Framework";
        System.Console.WriteLine(strDisplay);
    }
}
```

The *string* variable must be declared and set before it's used. This code is no good:

```
class Program
{
    static void Main()
    {
        System.Console.WriteLine(strDisplay);
        string strDisplay = "Hello, Microsoft .NET Framework";
    }
}
```

You'll get an compiler error message saying "The name 'strDisplay' does not exist in the current context." This code is no good either:

```
class Program
{
    static void Main()
    {
        string strDisplay;
        System.Console.WriteLine(strDisplay);
        strDisplay = "Hello, Microsoft .NET Framework";
    }
}
```

The variable is declared but it's uninitialized at the time *WriteLine* is called. The compiler error message is "Use of unassigned local variable 'strDisplay'."

The *strDisplay* variable is known as a *local* variable because it is declared within a method (in this case *Main*), and the variable is only visible within that method. You can also declare a variable outside of *Main* but within the class:

```
class Program
{
    static string strDisplay = "Hello, Microsoft .NET Framework";

    static void Main()
    {
        System.Console.WriteLine(strDisplay);
    }
}
```

The *strDisplay* variable is now known as a *field*, and it is potentially accessible to any method within the *Program* class. Both *strDisplay* and *Main* are considered members of the class. Notice that *strDisplay* is declared as *static*, meaning it is part of the class itself rather than an instance of the class. The program could refer to *strDisplay* by prefacing it with the class name:

```
System.Console.WriteLine(Program.strDisplay);
```

It doesn't matter where inside the class the *strDisplay* field is declared. This will work fine as well:

```
class Program
{
    static void Main()
    {
        System.Console.WriteLine(strDisplay);
    }

    static string strDisplay = "Hello, Microsoft .NET Framework";
}
```

This might look a little strange because in the context of the whole class *strDisplay* is declared after it's used, but that rule only applies to local variables. Both *Main* and *strDisplay* are members of the class, and the ordering of members usually doesn't matter. (However, if one field is set from the value of another field, then the ordering *does* matter.)

You can also declare a field but set its value in a method:

```
class Program
{
    static void Main()
    {
        strDisplay = "Hello, Microsoft .NET Framework";
        System.Console.WriteLine(strDisplay);
    }

    static string strDisplay;
}
```

If you leave out the assignment statement in *Main*, the program will still compile and run fine, but nothing will be displayed. If they're not explicitly initialized, fields are always implicitly initialized to zero values. A *string* field (and other reference types) is initialized to *null*.

But you can't have assignment statements outside of methods. This code doesn't compile at all:

```
class Program
{
    static string strDisplay;
    strDisplay = "Hello, Microsoft .NET Framework";

    static void Main()
    {
        System.Console.WriteLine(strDisplay);
    }
}
```

The compiler error message is "Invalid token '=' in class, struct, or interface member declaration," meaning that when the C# compiler was parsing the program, everything appeared OK until it got to the equal sign.

You can use the same name for fields and local variables:

```
class Program
{
    static string strDisplay = "This is a field";

    static void Main()
    {
        string strDisplay = "This is a local variable";
        System.Console.WriteLine(strDisplay);
    }
}
```

Within *Main*, the local variable takes precedence and the program will display "This is a local variable." However, because the field seems to serve no purpose in this program, the C# compiler will emit a warning message that says "The private field 'Program.strDisplay' is assigned but its value is never used."

That warning message suggests how you can access the field rather than the local variable:

```
class Program
{
    static string strDisplay = "This is a field";

    static void Main()
    {
        string strDisplay = "This is a local variable";
        System.Console.WriteLine(Program.strDisplay);
    }
```

```
}
```

Notice that *strDisplay* is now prefaced with the class name in the *WriteLine* call. The program displays "This is a field," But the compiler now complains with a warning message that "The variable 'strDisplay' is assigned but its value is never used."

If you look at the documentation for the *Console* class, and particularly the *WriteLine* method, you'll find lots of different versions. The one that we've been implicitly using is the one defined like this (in C# syntax):

```
public static void WriteLine(string value)
```

This method displays the string passed as an argument and then skips to the next line. The *void* keyword indicates that the method returns nothing to the caller. Exploring the *Console* class further, you'll also find a method named *Write*, and a version of the *Write* method defined like this:

```
public static void Write(string value)
```

The *Write* method displays its argument but does not skip to the next line. There's also a version of *WriteLine* that does nothing *but* skip to the next line:

```
public static void WriteLine()
```

There's no parameterless version of *Write* because it wouldn't do anything at all. You can rewrite the guts of FirstProgram so it looks like this:

```
Console.Write("Hello, ");
Console.Write("Microsoft ");
Console.Write(".NET ");
Console.Write("Framework!");
Console.WriteLine();
```

Notice that the first three strings end with a space so the words are still nicely separated.

If you look further in the *Console* documentation, you'll discover a method named *ReadLine*:

```
public static string ReadLine()
```

This method has no parameter, but it returns a *string*. This method obtains text typed by the user and then returns it to the program. You can store this return value in a string variable and then later display it.

GetTheUsersName.cs

```
//-----------------------------------------------
// GetTheUsersName.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class GetTheUsersName
{
```

```
    static void Main()
    {
        Console.Write("Type your name and press Enter: ");
        string strName = Console.ReadLine();
        Console.Write("Your name is ");
        Console.WriteLine(strName);
    }
}
```

Notice how the first *Console.Write* call is used to display the prompt. No new line is displayed and the cursor sits one space from the colon. The *Console.ReadLine* call echos typed characters to the console but does not return until the user presses Enter, which also causes the cursor to skip to the next line. The combination of *Console.Write* and *Console.WriteLine* then prints the information in a single line.

You can concatenate multiple strings using the plus operator, which means that those two last statements could have been written like this:

```
Console.WriteLine("Your name is " + strName);
```

A string literal must appear on a single line (except for verbatim strings, which can straddle multiple lines), so the concatenation operator is a good way to combine strings that are too long to fit comfortably on a single line.

Limerick.cs

```
//---------------------------------------
// Limerick.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;

class Limerick
{
    static void Main()
    {
        string strLimerick =
            "There once was a coder named Otto\r\n" +
            "Who had a peculiar motto:\r\n" +
            "    \"The goto is king,\r\n" +
            "    To thee I sing!\"\r\n" +
            "Maybe that's why he's often quite blotto.\r\n";

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

Notice the escape sequences for the embedded double quote marks in the third and fourth lines, and also that each of the five lines is terminated with escape sequences for a carriage return and line feed, which is the customary way to terminate lines in MS-DOS environments and Windows. Because the last line has a carriage return and line feed, and the entire string is displayed with *Console.WriteLine*, a blank line will appear after the end of the limerick.

In the documentation of the *Console* class, the *Write*, *WriteLine*, and *ReadLine* methods all appear in the section labeled "Methods." You'll also see a section labeled "Properties." If you have the SDK installed for the .NET Framework 1.0 or 1.1, you'll only see a few items under that heading. For versions 2.0 and above, however, you'll see a lot more. Let's examine a few of these items.

Here's how the property named *Title* is documented in C# syntax:

```
public static string Title { get; set; }
```

Like the methods in *Console*, this property is *public*, which means that we can access the *Title* property from outside the *Console* class, such as one of our programs. The property is also *static*, which means that we'll actually be referring to it as *Console.Title*. Each property has a type, and the type of this *Title* property is *string*. Within curly brackets appear the words *get* and *set*. This means that the property can be both read ("get") and set. When you write your own properties (which I'll get to in Chapter 17), you'll see how these words *get* and *set* figure in the property definition.

The *Console.Title* property is "gettable," which means that you can obtain and store the value of the property like this:

```
string strTitle = Console.Title;
```

Or, you can pass *Console.Title* to *WriteLine* to display the value of the property:

```
Console.WriteLine(Console.Title);
```

If you put this code at the top of Limerick.cs, it will display the same title as displayed in the titlebar of the console window in which Limerick runs.

The *Title* property is also "settable," which means you can put the following statement in Limerick.cs:

```
Console.Title = "Limerick";
```

This title will then appear at the top of the console window. (However, if you're compiling and running on the command line, the title will only be changed for the duration the program is running, which is a very short time. You might want to put a *Console.ReadLine* call at the bottom of the program to actually see the new title take effect.)

As you can see, the syntax involved in getting and setting *Title* makes it look like a field. But it's not that simple. Although properties certainly resemble fields in use, properties are actually implemented with code. There is actual code being executed when you obtain or set a property.

If you insert statements to access and change *Title* in the Limerick program, and then you look at the executable with the IL Disassembler, you'll see that *Title* has magically changed to the method calls *get_Title*

and *set_Title*. Although properties have the syntax of fields, they are implemented in the *Console* class as methods.

The *Console* class also defines properties named *BackgroundColor* and *ForegroundColor*. These two properties are also gettable and settable, but the type of the properties is *ConsoleColor*. What on earth is *ConsoleColor*? If you look a little further in the *System* namespace documention, you will indeed see a page entitled "ConsoleColor Enumeration."

ConsoleColor is an enumeration, which means that it has a collection of members that have been associated with integers. In C#, enumerations are strongly typed, and the enumeration member has to be prefaced with the enumeration name. Here's how you set the *Background* and *Foreground* colors in a program:

```
Console.BackgroundColor = ConsoleColor.Yellow;
Console.ForegroundColor = ConsoleColor.Blue;
```

Using enumerations in C# involves a little more typing than you may be accustomed to, but there is very little room for confusion or blunders.

If you put those *Background* and *Foreground* statements at the top of Limerick.cs, the results won't be very attractive because only the characters displayed by the program will be rendered with these new colors. After setting the colors, you probably want to clear the console screen with a call to the static method:

```
Console.Clear();
```

The Limerick.cs file terminates every line with the characters '\r' and '\n', which denote a carriage return and line feed. A '\n' works by itself to skip to the next line, but a '\r' by itself causes the next line to overwrite the terminated line. As you might know, the next-line character varies by the operating system platform, and if you really want to help your programs achieve some kind of platform independence, you might consider using instead the static property *Environment.NewLine*. That's the static *NewLine* property in the *Environment* class, which is also part of the *System* namespace. This property is intended to be appropriate for the particular environment on which the program is running.

The *Environment.NewLine* property is documented in C# syntax like this:

```
public static string NewLine { get; }
```

The type is *string*, but the property is get only. You cannot set the *Environment.NewLine* property. Here's how you might use it in Limerick.cs:

```
"There once was a coder named Otto" + Environment.NewLine +
```

That's not exactly a pleasant solution if you have to repeat it five times. Instead, you might begin by defining a local variable just for the new-line sequence:

```
string NL = Environment.NewLine;
```

Or you might define *NL* as a field if you need it in more than one method. Either way you can then refer to *NL* at the end of each line:

```
"There once was a coder named Otto" + NL +
```

Much better!

C# also defines a *char* type for storing a single 16-bit Unicode character. But strictly speaking, *char* is not a numeric type as it is in C and C++. There is no such thing as a signed *char* or an unsigned *char*. A character literal is defined with single quotation marks, and all the escape sequences shown earlier in this chapter are valid:

```
char chBackSlash = '\\';
```

You can concatenate *char* variables with *string* variables or literal strings:

```
string strDirectory = "C:" + chBackSlash + "Windows";
```

You can also concatenate strings and integers. Here's an example:

```
Console.WriteLine("Schubert was " + 31 + " when he died.");
```

That will display as:

```
Schubert was 31 when he died.
```

What's going on behind the scenes here is actually a bit more involved than you might imagine, but I don't want to give away the secret just yet.

As in C and C++, the basic integral data type in C# is the *int*. You can declare and initialize an *int* like this:

```
int age = 31;
```

You can then concatenate that variable with the string:

```
Console.WriteLine("Schubert was " + age + " when he died.");
```

The result is the same as before. You might try performing a calculation in the *Console.WriteLine* statement using the years in which Schubert was born and died:

```
Console.WriteLine("Schubert was " + 1828 - 1797 + " when he died.");
```

This will *not* work. C# (like C and C++) evaluates additive operators like plus and minus from left to right. The first plus sign causes the number 1828 to be concatenated to the string "Schubert was " and the result is "Schubert was 1828". Then there's a minus sign, and that's a problem because we now have a string minus a number.

A simple set of parentheses around the calculation will help:

```
Console.WriteLine("Schubert was " + (1828 – 1797) + " when he died.");
```

The subtraction is now performed first and the result is concatenated with the strings. You can even start with the number, as in this variation in Yoda syntax:

```
Console.WriteLine(31 + " when he died Schubert was.");
```

This will work as well:

```
Console.WriteLine(1828 – 1797 + " when he died Schubert was.");
```

As in C and C++, the standard floating-point data type in C# is the *double*. Here's a declared and initialized *double*, and a statement that displays the value:

```
double onethird = 1 / 3.0;
Console.WriteLine("One divided by three equals " + onethird);
```

As you'll note, the expression that's set to the variable *onethird* is *not* written as 1 divided by 3. Like C and C++, C# interprets numeric literals without decimal points as integers, and integer division is performed with truncation, so the result would be zero. Expressing one of the two numbers as a floating-point literal causes the other to be converted to floating point for the division. The *WriteLine* statement displays:

```
One divided by three equals 0.333333333333333
```

The *Math* class is the *System* namespace contains a collection of static methods that mostly perform logarithmic and trigonometric calculations. The *Math* class also contains two constant fields named *PI* and *E*, which are of type *double*. Here's a statement using *Math.PI*:

```
Console.WriteLine("A circle's circumference divided by its diameter is "
                  + Math.PI);
```

That statement displays:

```
A circle's circumference divided by its diameter is 3.14159265358979
```

I'm sure that some programmers want to know: How does C# store strings? Are strings terminated with zero characters as they are in C and C++, or something else? And the answer is: Something else.

The *string* keyword is C# is actually an alias for a class in the *System* namespace named *String*. Notice the difference in case: The C# keyword is *string* but the class is *String*. In any C# program, you can replace the word *string* with *System.String* and the program will be exactly the same:

```
System.String str = "Just a string";
```

If you have a *using* directive for the *System* namespace, you can replace *string* with *String* and use the two forms interchangeably:

```
String str = "Another string";
```

What you *cannot* do is refer to the *System.string* class (notice the lowercase *string*):

```
System.string str = "Not a workable string";    // Won't work!
```

Because *string* is an alias for *System.String*, that translates as *System.System.String*, which does not exist.

Similarly, the *char* data type is an alias for the structure *System.Char*, and the *int* data type is an alias for the structure *System.Int32*, and *double* is an alias for *System.Double*. As Jeffrey Richter points out (*CLR via C#*, page 119), it's as if every C# program contained *using* directives like the following:

```
using string = System.String;
using char = System.Char;
using int = System.Int32;
using double = System.Double;
```

And so forth. (There are more basic data types than just these four.)

The more profound repercussion is this: Any *string* variable can also be termed "an object of type *String*" or "an instance of the *String* class." And the *String* class itself provides many goodies. As you explore the documentation of the *String* class you'll discover many methods with quasi-familiar names: *Substring*, *LastIndexOf*, *ToLower*, *ToUpper*, and many more. All these methods perform various manipulations of strings.

The *String* class also has two important properties. The *Length* property is defined like so:

```
public int Length { get; }
```

This property is of type *int*, and it is get-only. But the big difference compared with the other properties you've seen so far is the absence of the *static* keyword. *Length* is not a static property of the *String* class. *Length* is, instead, an *instance* property, which means that it applies to a particular *string* variable rather than to the *String* class. In the big scheme of things, instance properties (and instance methods) are much more common than static properties and static methods—so much so that properties and methods are instance by default.

You don't preface the *Length* property with the *String* class name. You don't use the expression *String.Length*. What could that possibly mean? Instead, you use the *Length* property with an instance of the *String* class—what we have been casually calling a *string* variable:

```
string strMyString = "This is my string";
Console.WriteLine(strMyString.Length);
```

The expression *strMyString.Length* returns the length of the string, in this case the number 17.

Length does not return information about the ice cream dispenser that is the *String* class. *Length* measures the size of an individual sundae.

You can also apply the *Length* property to a string literal:

```
"This is a string literal".Length
```

That expression has a value of 24.

The *String* class has two properties, and the other property seems to be named *Chars*. Indeed, in some programming languages you might actually use that property name. However, in the C# representation of the *Chars* property declaration, you'll see the following:

```
public char this [int index] { get; }
```

The word *Chars* does not appear in this declaration. Instead, we see a property that seems to have a name of *this*, and the property is of type *char*. But *this* is actually a C# keyword, and in this context it's a rather special syntax. This declaration defines an *indexer* for the *String* class, which indicates that you can use square brackets to index a string variable and obtain its characters. For example, the expression

```
strMyString[0]
```

returns the first character of the *strMyString*, which, as defined above is the character 'T'.

The syntax is the same as indexing a C or C++ array (and actually the same as indexing an array in C#). Indexing begins at 0, so the expression *strMyString[5]* is the 6th character of the string, or 'i'. You can also index string literals:

```
"This is a string literal"[15]
```

That's the character 'g'. The index can range from 0 to one less than the *Length* property of the string. Here's a little program that demonstrates the *Length* and indexer properties.

StringProperties.cs

```
//-----------------------------------------------
// StringProperties.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class StringProperties
{
    static void Main()
    {
        Console.Write("Enter some text: ");
        string strEntered = Console.ReadLine();
        Console.WriteLine();
        Console.WriteLine("The text you entered has " + strEntered.Length +
                          " characters");
        Console.WriteLine("The first character is " + strEntered[0]);
        Console.WriteLine("The last character is " +
                                    strEntered[strEntered.Length - 1]);
        Console.WriteLine();
    }
}
```

Of course, because you are an inquisitive person, you'll want to see what happens when you press the Enter key without typing any text at all. The

Console.ReadLine method returns an empty string in that case, the *Length* property of an empty string is 0, but the first indexer, which is *strEntered[0]*, has a little problem because there's no character for it to return. You'll probably get a dialog box informing you of a problem, and then some text in the console window that begins like this:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside
the bounds of the array.
```

The fully-qualified StringProperties.cs filename will also be displayed in this message, and a line number where the problem occurred. That line number might be exact if you're compiling in Debug mode, or it might only refer to the method in which it occurred.

At any rate, the program gracefully terminated. It didn't hang, or display a bunch of funny characters to the screen, or bring down Windows along with it. Notice that the message says "Unhandled Exception," which implies that there's a way for you, the programmer, to write code that handles this problem without terminating the program. (You'll see how to handle exceptions in Chapter 12.) *IndexOutOfRangeException* is a class in the *System* namespace, and it's one of several classes for various types of exceptions a program might encounter. I'll be using these class names to refer to common exceptions.

The exception message indicates that the "Index was outside the bounds of the array," and that might prompt you to ask: Is a *string* really just an array of characters? Definitely not in the same sense that C and C++ strings are arrays of characters. The big difference is implied in the definition of the indexer:

```
public char this [int index] { get; }
```

This property is get-only. Code like this is simply not allowed:

```
strMyString[5] = 'a';    // Can't do it!
```

Once a string is created, you can't change the individual characters. Nor is there any method in the *String* class that can change the characters of the string. The string is said to be *immutable*.

What does this mean? Well, what it certainly does *not* imply is a prohibition against setting a string variable to another string. This code is perfectly legal:

```
string str = "first string";
str = "second string";
```

But that's really the *only* way you can change the contents of a *string* variable. You have to set the variable to a different string rather than changing the characters of the existing string.

Because strings are immutable, certain operations common in C and C++ are no longer possible in C#. For example, Microsoft C and C++ include a

library function named *_strupr* that converts a string to uppercase. In a C program, if *pMyCString* is a pointer to *char* or an array of *char*, you'd use *_strupr* like this:

```
_strupr(pMyCString);
```

The *_strupr* function takes each character in *pMyCString* and converts it to upper-case and stores it back in the same place. The *_strupr* function returns a pointer to the converted string, but it's the same pointer passed to the function.

The equivalent method of the *String* class is *ToUpper*. But for a string instance named *strMyCSharpString* you can't just call the method like so:

```
strMyCSharpString.ToUpper();    // Won't do anything!
```

Syntactically, this statement is valid, but it has no effect on the *strMyCSharpString* variable. Strings are immutable and hence the characters of *strMyCSharpString* can't be altered. The *ToUpper* method actually creates a new string. You need to assign the return value of *ToUpper* to a string variable:

```
string strMyUpperCaseString = strMyCSharpString.ToUpper();
```

Or you could assign the return value to the same string variable:

```
strMyCSharpString = strMyCSharpString.ToUpper();
```

In the second case, the original string (the one containing lowercase letters) still exists, but since it's probably no longer referenced anywhere in the program, it becomes eligible for garbage collection.

So, suppose you have a string defined like so

```
string str = "abcdifg";
```

and you want to change the fifth character to an 'e'. You know you can't do it this way:

```
str[4] = 'e';    // Won't work!
```

The indexer is get-only. So how *do* you do it? There are a couple possible approaches, which you can dig up by searching through the *String* documentation. The method call

```
str = str.Replace('i', 'e');
```

returns a string where *all* the occurrences of 'i' have been replaced with 'e' . Notice the return value from *Replace* is assigned to the same varaiable that went into the method. Alternatively, you can first call *Remove* to create a new string with one or more characters removed at a specified index with a specified length. For example, the call

```
str = str.Remove(4, 1);
```

removes one character at the fourth position (the 'i'). You can then call *Insert* to insert a new string, which in this case is a single character:

```
str = str.Insert(4, "e");
```

Or you can do both jobs in one statement:

```
str = str.Remove(4, 1).Insert(4, "e");
```

Despite the use of a single *string* variable named *str*, the two method calls in this last statement create two additional strings, and the quoted "e" is yet another string.

Or you can patch together a new string from substrings:

```
str = str.Substring(0, 4) + "e" + str.Substring(5);
```

Now that I have you looking through the documentation for the *String* class, you'll notice it includes a section labeled "String Constructor." Constructors serve to create and initialize objects. In use, a constructor requires the keyword *new* followed by the class name itself and possible arguments in parentheses. Here's a declaration of a *string* variable that uses one of the constructors defined by the *String* class:

```
string strAtSigns = new string('@', 15);
```

You can use either the lowercase *string* keyword or the uppercase *String* class when calling a constructor. This particular constructor is defined with a *char* as the first parameter and an *int* as the second parameter. It creates a string containing 15 occurrences of the @ character, which is enough for a charm bracelet:

```
@@@@@@@@@@@@@@@
```

Many of the string constructors create strings based on pointers, but one handy one creates a string from an array of characters. This constructor suggests yet another approach to replacing a character in a particular string. You can convert the string into a character array, set the appropriate element of the array, and then construct a new string based on the character array. In C#, array variables are declared by following the type of the array with empty double brackets:

```
char[] buffer = str.ToCharArray();
buffer[4] = 'e';
str = new string(buffer);
```

That array syntax may seem a little strange to the C and C++ programmer, but I'll discuss it in detail in Chapter 10.

As I've mentioned, just as *string* is an alias for the *System.String* class, the *char*, *int*, and *double* keywords in C# are also aliases. But these are not aliases for classes in the *System* namespace. Instead, they're aliases for *structures*.

The difference between classes and structures will become more apparent in later chapters. But in many ways classes and structures are similar, and instead of saying "this is my *int* variable," you can instead say "this is an instance of the *Int32* structure" or "this is an object of type

Int32" or "this is an *Int32* object." It makes the humble thing sound just a little bit more important.

Although you won't see any constructors listed in the documentation for the *Int32* and *Double* structures, both structures have default parameterless constructors that return zero values of the object. So, instead of initializing an *int* like this

```
int index = 0;
```

you can do it like this:

```
int index = new int();
```

Or this:

```
System.Int32 index = new int();
```

Or this:

```
int index = new System.Int32();
```

Or, if you have a *using* directive for the *System* namespace, like this:

```
Int32 index = new Int32();
```

Or any other combination. All those declarations are equivalent. Beginning in .NET 2.0, you can also use the *default* keyword to obtain the default value of an *int*:

```
int index = default(int);
```

As you saw in the documentation of the *String* class, there seems to be lots of good reasons why the *string* data type is an alias for a class. But is there any reason that the *int* and *double* types are aliases for the *Int32* and *Double* structures?

Oh, yes. Both structures have instance methods named *ToString* that convert the object to a string. In fact, every single class and structure in the .NET Framework—including those that you will create yourself—has an instance method named *ToString*. This is because the *System.Object* class (also known by its C# alias *object*) defines a method named *ToString*. The *System.Object* class is the grand matriarch of every .NET class and structure, and they all inherit this wonderful *ToString* method, and many classes and structures tailor *ToString* to their own requirements.

Consider the following code:

```
int i = 55;
string str = i.ToString();
```

The *ToString* method converts the *int* variable to a *string*, which in this case is the string "55". You can even apply *ToString* to a integer literal:

```
12345.ToString()
```

That *ToString* call returns the string "12345".

And this is how the concatenation of *string* objects with non-*string* objects works. If a *string* variable or literal is on either side of a plus sign, and if a non-*string* is on the other side, then the non-*string* object is converted to a *string* by a call to its *ToString* method. It works every time!

Earlier I showed how to use the static *NewLine* property from the *Environment* class. The *Environment* class has some other goodies that can give your program information about the machine on which it's running. Here's a program that shows just a couple of these items.

ShowEnvironmentStuff.cs

```
//-----------------------------------------------------
// ShowEnvironmentStuff.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;

class ShowEnvironmentStuff
{
    static void Main()
    {
        Console.WriteLine("My Documents is actually " +
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments));
        Console.WriteLine();

        int msec = Environment.TickCount;
        Console.WriteLine("Windows has been running for " +
                          msec + " milliseconds");
        Console.WriteLine("\tor " + msec / 3600000.0 + " hours");
        Console.WriteLine();

        Console.WriteLine("You are running " + Environment.OSVersion);
        Console.WriteLine("\tand .NET version " + Environment.Version);
        Console.WriteLine();
    }
}
```

The program begins by obtaining the actual directory referred to as *My Documents*. This is available through a static *Environment.GetFolderPath* method, which returns a *string*. The argument is a member of the *SpecialFolder* enumeration, but the *SpecialFolder* enumeration is actually defined within the *Environment* class, which is why the lengthy member name is *Environment.SpecialFolder.MyDocuments*.

The static *Environment.TickCount* property returns an *int* indicating the the number of milliseconds that the current Windows session has been running. One *WriteLine* call in the program displays this value directly while the other divides it by 3600000.0 to get the floating-point value in hours.

When I ran this program under Windows XP, the first two sections of the program displayed the following information:

```
My Documents is actually C:\Documents and Settings\Charles\My Documents
```

```
Windows has been running for 16729593 milliseconds
        or 4.64710916666667 hours
```

When I ran the program under a Windows Vista partition, the first item displayed as:

```
My Documents is actually H:\Users\Charles\Documents
```

The last section of the program displays two static properties named *Environment.OSVersion* and *Environment.Version*. The *Version* property is documented as returning an object of type *Version*, which is a class defined in the *System* namespace. It may be a bit confusing that a property named *Version* returns an object of type *Version*, but that naming convention is quite common in .NET.

The *Version* class (also defined in the *System* namespace) has four crucial *int* properties named *Major*, *Minor*, *Build*, and *Revision*. The *ToString* method defined by the *Version* class nicely displays these four numbers separated by periods, just as we've come to expect version numbers to be displayed.

The *Environment.OSVersion* property returns an object of type *OperatingSystem*. The crucial properties of the *OperatingSystem* class are *Platform* (which is a member of the *PlatformID* enumeration), *Version* (which is an object of type *Version*), and *ServicePack*, which is a string. Again, the *ToString* method nicely renders this information in a readable form, so that the last section of the ShowEnvironmentStuff program displays the information (when I ran it under Windows XP):

```
You are running Microsoft Windows NT 5.1.2600 Service Pack 2
        and .NET version 2.0.50727.42
```

Under Windows Vista, the program reported:

```
You are running Microsoft Windows NT 6.0.6000
        and .NET version 2.0.50727.312
```

If you want to be a little more explicit about the objects returned from the *OSVersion* and *Version* properties, you can do this:

```
Version vers = Environment.Version;
OperatingSystem opsys = Environment.OSVersion;
Console.WriteLine("You are running " + opsys);
Console.WriteLine("\tand .NET version " + vers);
```

Here, *vers* is declared as an object of type *Version*, and *opsys* is declared as an object of type *OperatingSystem*, which are the types of the objects returned from *Envionment.Version* and *Environment.OSVersion*.

Learning about C# generally begins with the static *Main* method and static methods of the *Console* class, but static methods and properties are generally the exception rather than the rule. In general, a program deals with instances of classes and structures.

The only reason *Console* is entirely static is because to any application, there is only one *Console*. If an application could create multiple consoles, then the *Console* class would have a constructor that returned an instance of the *Console* class, and *WriteLine* would be an instance method. You'd precede *WriteLine* with one of the instances of the *Console* class to indicate on which console you want to display the text.

Environment, also, is a collection of static methods and properties because to any application, there is only one operating system environment and machine on which it's running. (However, the properties of *Environment* return instances of other classes.)

In .NET 1.0, it was actually possible to create instances of the *Console* and *Environment* classes using a *new* expression like this:

```
Console cons = new Console();    // Doesn't work any more.
```

But that no longer works. Both *Console* and *Environment* contain nothing but static methods and properties so the class definitions themselves also contain the *static* keyword:

```
public static class Console
{
    ...
}
```

Because you can't create an instance of the *Console* class, you can't call the *ToString* method in *Console* because *ToString* is always an instance method.

Although the *Int32* and *Double* structures are primarily for working with *int*, and *double* instances, these structures also have some static members. In particular, these structures have static methods named *Parse* that convert strings into numbers.

The static *Int32.Parse* method accepts a *string* argument and returns an object of type *Int32*. The *Parse* method is pretty much the opposite of *ToString*. As you know by now, *ToString* is an instance method because it applies to a particular integer. You must have an integer—either a variable or a literal or perhaps the return value of a method—to call the *ToString* method of the *Int32* struture:

```
int i = 275;
string str = i.ToString();
```

Int32.Parse is a static method. You use this method by specifying the *Int32* structure to the left of the method name. You don't need to have an integer around to call *Int32.Parse*. The method creates an integer for you:

```
string str = "275";
int i = Int32.Parse(str);
```

Because *int* is an alias for *System.Int32*, you can actually call *Parse* like this:

```
int i = int.Parse(str);
```

Although it's perfectly legal, it sure looks peculiar. I prefer using the actual class or structure name when calling static methods.

The *System.Double* structure has a static method named *Parse* as well, and the *Parse* methods in both structures have overloads that accept a member of the *NumberStyles* enumeration to govern the type of input *Parse* will accept.

Here's a program that uses *Double.Parse* with a static method from the all-static *Math* class to calculate powers.

Exponentiation.cs

```
//-----------------------------------------------
// Exponentiation.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class Exponentiation
{
    static void Main()
    {
        Console.Write("Enter the base: ");
        double number = Double.Parse(Console.ReadLine());

        Console.Write("Enter the power: ");
        double power = Double.Parse(Console.ReadLine());

        Console.WriteLine(number + " to the " + power + " power equals " +
                          Math.Pow(number, power));
    }
}
```

Notice that in both cases the argument to *Double.Parse* is a call to *Console.ReadLine*, which returns a *string* that is then passed to the *Parse* method.

If you type something in that *Parse* can't handle, you'll get a *FormatException*. You can either trap that exception, or you can use an alternative method named *TryParse* that doesn't raise an exception for improper input. (I'll describe both approaches in Chapter 12.)

Some of the common classes and structures defined in the .NET *Framework* define multiple versions of the *ToString* method. The *Double* structure, for example, defines four different *ToString* methods:

```
string ToString()
string ToString(string format)
string ToString(IFormatProvider provider)
string ToString(string format, IFormatProvider provider)
```

The second version of the *ToString* method allows you to use a formatting string that consists of a letter optionally followed by a number. For example, if *num* is a variable of type *double*, then

```
num.ToString("F3");
```

displays *num* in "fixed-point" style with three decimal places. The following program demonstrates some of the options you have in displaying numbers.

NumericFormatting.cs

```
//-----------------------------------------------
// NumericFormatting.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class NumericFormatting
{
    static void Main()
    {
        Console.WriteLine("Currency    C3: " + Math.PI.ToString("C3"));
        Console.WriteLine("Exponential E3: " + Math.PI.ToString("E3"));
        Console.WriteLine("Fixed-Point F3: " + Math.PI.ToString("F3"));
        Console.WriteLine("General     G3: " + Math.PI.ToString("G3"));
        Console.WriteLine("Number      N3: " + Math.PI.ToString("N3"));
        Console.WriteLine("Percent     P3: " + Math.PI.ToString("P3"));
        Console.WriteLine("Round-Trip  R3: " + Math.PI.ToString("R3"));
        Console.WriteLine();
        Console.WriteLine("Fixed-Point F3: " + 12345678.9.ToString("F3"));
        Console.WriteLine("General     G3: " + 12345678.9.ToString("G3"));
        Console.WriteLine("Number      N3: " + 12345678.9.ToString("N3"));
        Console.WriteLine();
        Console.WriteLine("Decimal     D3: " + 55.ToString("D3"));
        Console.WriteLine("General     G3: " + 55.ToString("G3"));
        Console.WriteLine("Hexadecimal X3: " + 55.ToString("X3"));
    }
}
```

The first seven statements display π and the following results appear when you run this program with your region set to the United States:

```
Currency    C3: $3.142
Exponential E3: 3.142E+000
Fixed-Point F3: 3.142
General     G3: 3.14
Number      N3: 3.142
Percent     P3: 314.159 %
Round-Trip  R3: 3.1415926535897931
```

In many cases, the number you provide in the formatting string indicates the number of decimal places. The exception is General formatting, in which case the number indicates the total number of digits displayed. General formatting will use either Exponential or Fixed-Point formatting, depending which one is most economical for the particular number. If you use the parameterless version of *ToString*, it is equivalent to "G".

As you can see, the Round-Trip formatting ignores the number in the formatting string, and creates a string that can be passed to *Parse* to get the original number back.

The next three statements in the program show what happens with a number with more digits to the left of the decimal point:

```
Fixed-Point F3: 12345678.900
General     G3: 1.23E+07
Number      N3: 12,345,678.900
```

In this case, General formatting uses the Exponential format because it's more economical. The Number formatting string inserts commas as thousands separators (or whatever is regionally appropriate).

Two of the formatting options—Decimal and Hexadecimal—are for use only with integers. Both insert zeros to the left of the number if necessary to make it as wide as the number you specify in the formatting string:

```
Decimal     D3: 055
General     G3: 55
Hexadecimal X3: 037
```

If you change the number following the D or X to 1, you'll probably be happy to note that no actual digits are stripped from the result.

If you go into the Control Panel and invoke the Regional and Language Options applet, you can change some settings—for example, the currency symbol and the thousands separator—that affect how *ToString* displays the number. By default, *ToString* uses the regional settings, but this behavior may be undesirable in some cases. You might want to display currency in dollars or euros *regardless* of the user's regional settings.

This option is made possible by the third and fourth overloads of the *ToString* method, which have the following syntax:

```
string ToString(IFormatProvider provider)
string ToString(string format, IFormatProvider provider)
```

If you look in the documentation of the *System* namespace, you'll find *IFormatProvider* is identified as an *interface*. (By convention, all interfaces in the .NET Framework begin with the capital letter I.) You'll also see that *IFormatProvider* has one method defined, which is named *GetFormat*.

Interfaces contain no code of their own. Somewhere within the source code for the .NET Framework, *IFormatProvider* is probably defined in its entirety like this:

```
public interface IFormatProvider
{
    object GetFormat(Type formatType);
}
```

What actually must be passed to the *ToString* method is an instance of a class that *implements* the *IFormatProvider* interface, and by that it is meant that the class contains a method named *GetFormat* defined in the

same way as the signature in *IFormatProvider*, but which has actual code.

For the job of formatting numbers, the relevant class that implements the *IFormatProvider* interface is named *NumberFormatInfo*, and it's defined in the *System.Globalization* namespace. To refer to the *NumberFormatInfo* class in your program, you'll either need to preface the class name with *System.Globalization*, or provide a *using* directive for *System.Globalization*. (I'll assume the latter.)

To customize the formatting of numbers by *ToString*, you need an instance of the *NumberFormatInfo* class. If you look at the properties of this class, you'll see stuff like *CurrencySymbol*, *CurrencyDecimalSeparator*, and *PercentSymbol*. All these properties are settable and gettable. This means that you can create an object of type *NumberFormatInfo*, set the properties to whatever you want, and then pass that object to *ToString* to get the desired result.

You can create an object of type *NumberFormatInfo* using the parameterless constructor defined by the class:

```
NumberFormatInfo info = new NumberFormatInfo();
```

You can then set some of the properties like so:

```
info.CurrencySymbol = "\x20AC";
info.CurrencyPositivePattern = 3;
info.CurrencyNegativePattern = 8;
```

The Unicode character 0x20AC is the symbol for the Euro. If you look at the documentation of *CurrencyPositivePattern*, you'll see that a value of 3 means that the currency symbol is to be displayed after the number and separated by a space. A *CurrencyNegativePattern* of 8 means that the symbol will appear in the same place for negative numbers, but a negative sign will appear in front of the number.

There are other ways to get instances of the *NumberFormatInfo* class without explicitly using the constructor. The *NumberFormatInfo* class has two static methods named *CurrentInfo* and *InvariantInfo* that return instances of the *NumberFormatInfo* class.

It might sound a little peculiar that static methods of a class return instances of the class, but it's perfectly legitimate. Here's what the syntax might look like:

```
NumberFormatInfo info = NumberFormatInfo.CurrentInfo;
```

CurrentInfo is a static property of the *NumberFormatInfo* class, so it must be prefaced with the name of the class. It is get-only, and it returns an object of type *NumberFormatInfo*, which you can then save in a variable of type *NumberFormatInfo*. The code for the *CurrentInfo* property obviously invokes the *NumberFormatInfo* constructor to create an

instance of the class. Then it sets a bunch of instance properties on the object.

The static *NumberFormatInfo.CurrentInfo* property returns a *NumberFormatInfo* object with settings that are applicable for your particular culture as you've indicated in the Control Panel.

Similarly, the static *NumberFormatInfo.InvariantInfo* property returns an instance of *NumberFormatInfo* initialized with "invariant" settings, that is, settings that do not depend on any particular culture, and which will be the same regardless of the machine on which the program is running.

The *NumberFormatInfo* constructor creates a *NumberFormatInfo* instance initialized with invariant information. It is quite likely that the static *NumberFormatInfo.InvariantInfo* property is implemented by a simple call to the constructor.

If you want to call *ToString* with an object of type *NumberFormatInfo*, you can simply pass *NumberFormatInfo.InvariantInfo* or *NumberFormatInfo.CurrentInfo* directly to the *ToString* method. (If you use a simpler form of *ToString*, *NumberFormatInfo.CurrentInfo* is used by default.)

The following program displays a number in a currency format with *NumberFormatInfo.InvarientInfo* (which uses a currency symbol of ¤), *NumberFormatInfo.CurrentInfo* (which will use the currency symbol you've indicated in the Control Panel), and a custom version that displays the Euro symbol (which, unfortunately, does not render on the console and shows up as a question mark).

CurrencyFormatting.cs

```
//-----------------------------------------------
// CurrencyFormatting.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Globalization;

class CurrencyFormatting
{
    static void Main()
    {
        double money = 1234567.89;

        Console.WriteLine("InvariantInfo: " +
            money.ToString("C", NumberFormatInfo.InvariantInfo));

        Console.WriteLine("CurrentInfo:   " +
            money.ToString("C", NumberFormatInfo.CurrentInfo));

        NumberFormatInfo info = new NumberFormatInfo();
        info.CurrencySymbol = "\x20AC";
        info.CurrencyPositivePattern = 3;
        info.CurrencyNegativePattern = 8;
```

```
        Console.WriteLine("Custom Info:   " + money.ToString("C", info));
    }
}
```

Notice the *using* directive for the *System.Globalization* namespace.

I know you really, really want to see the Euro symbol, so let's put this basic logic into a small Windows Forms program. For this next project, you'll need to add a reference of System.Windows.Forms.dll.

CurrencyFormattingMessageBox.cs

```
//-----------------------------------------------------------
// CurrencyFormattingMessageBox.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Globalization;
using System.Windows.Forms;

class CurrencyFormatting
{
    static void Main()
    {
        double money = 1234567.89;
        string strDisplay;

        strDisplay = "InvariantInfo: " +
            money.ToString("C", NumberFormatInfo.InvariantInfo) +
            Environment.NewLine;

        strDisplay += "CurrentInfo: " +
            money.ToString("C", NumberFormatInfo.CurrentInfo) +
            Environment.NewLine;

        NumberFormatInfo info = new NumberFormatInfo();
        info.CurrencySymbol = "\x20AC";
        info.CurrencyPositivePattern = 3;
        info.CurrencyNegativePattern = 8;

        strDisplay += "Custom Info: " + money.ToString("C", info);

        MessageBox.Show(strDisplay, "Currency Formatting");
    }
}
```

Notice the *using* directive for *System.Windows.Forms*. In this program, the *strDisplay* variable appears three times on the left of assignment and compound assignment statements, and becomes the first argument to the *MessageBox.Show* static method.

ToString is certainly a powerful tool for formatting. But in general practice a somewhat different approach is used that is very similar to the C and C++ *printf* function. The first argument to *Console.WriteLine* is a formatting string, and subsequent arguments are the items to be displayed. For example:

```
Console.WriteLine("{0} times {1} equals {2}", A, B, A * B);
```

In the formatting string, the numbers surrounded by curly brackets are placeholders. The numbers correspond to the subsequent arguments, where 0 is the first of the subsequent arguments, 1 is the second, and so forth. The placeholder numbers in the formatting string don't have to be in numeric order:

```
Console.WriteLine("{2} equals {0} times {1}", A, B, A * B);
```

You can repeat a particular placeholder number:

```
Console.WriteLine("{0} times {0} equals {1}", A, A * A);
```

You can skip placeholder numbers. In the following case, the variables *C*, *D*, and *E* are ignored and don't get displayed:

```
Console.WriteLine("{1} times {2} equals {4}", C, A, B, D, A * B, E);
```

The only restriction is that the number of subsequent arguments must be greater than the highest placeholder number.

If you need to display curly braces within the formatting string, use a pair of curly braces: {{ or }} will be interpreted as a displayable symbol.

To control the formatting of the individual arguments, you follow the placeholder number with a colon and the same formatting string you would pass to the *ToString* method:

```
Console.WriteLine("{0:N2} times {1:N2} equals {2:N4}", A, B, A * B);
```

Basically, this statement is equivalent to:

```
Console.WriteLine(A.ToString("N2") + " times " + B.ToString("N2") +
                  " equals " + (A * B).ToString("N4"));
```

You can also control the width of the field allowed for the string returned from *ToString*. You do this by following the placeholder number with a comma and a field width in characters:

```
Console.WriteLine("{0,5:N2} times {1,5:N2} equals {2,10:N4}",
                  A, B, A * B);
```

What you're indicating here is a *minimum* field width. The method won't truncate the result of *ToString*. In this example, A and B will be displayed in a field of at least 5 characters width, and the product with 10 characters width. The number is right justified in the field, and spaces are used to pad to the left. For left justification, use a negative field width:

```
Console.WriteLine("{0,-5:N2} times {1,-5:N2} equals {2,-10:N4}",
                  A, B, A * B);
```

These field widths are useful for aligning numbers in columns when displaying multiple rows.

As you've seen, the *ToString* methods of the individual classes and structures are responsible for the bulk of the formatting job. The rest of

it isn't really handled by *Console.WriteLine*. When you use a formatting string, *Console.WriteLine* just hands the job over to another static method named *String.Format*. If *Console.WriteLine* is the C# version of *printf*, then *String.Format* is the C# version of *sprintf*.

So, even if you never write another console program for the rest of your life, you probably will still make use of *String.Format* for formatting objects for display.

Chapter 6. Primitive Data Types

As you've seen C# supports data types of *string*, *char*, *int*, and *double*. This chapter takes a more methodical approach to these and the other primitive data types supported by C# and the CLR.

Integers

Like C and C++, C# supports integral data types named *short*, *int*, and *long*. But almost immediately, the differences between C/C++ and C# start to become apparent. The C and C++ standards impose *minimum* bit widths for the three integral data types, but C# fixes them precisely. The C# *short* is always 16 bits wide, the *int* is 32 bits wide, and the *long* is 64 bits wide.

The *short*, *int*, and *long* are signed integers stored in standard two's complement format. The *short* can range in value from -32,768 through 32,767, the *int* from -2,147,483,648 through 2,147,483,647, and the long from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Within a method like *Main*, you can declare an *int* named *i* (for example) with the declaration statement:

```
int i;
```

This statement essentially allocates 4 bytes of space on the stack to store the value of *i*. However, at this point, *i* is uninitialized, and any attempt to refer to its value will provoke the C# compiler to report a problem. For example, the C# compiler won't allow this code:

```
int i;
Console.WriteLine(i);
```

After *i* is declared, you can assign it a value:

```
i = 545;
```

Or, you can initialize *i* in its declaration statement:

```
int i = 545;
```

In either case, the number 545 is an integer literal that is normally assumed to be of type *int*, except that the C# compiler will bend the rules a bit if you assign the literal to a *short*, for example:

```
short s = 545;
```

You can assign the value of an *int* to a *long*:

```
long l = i;
```

This is known as an *implicit* cast. The *int* is converted to a *long* without difficulty. Similarly, you can assign a *short* to an *int* or a *long* without a cast. However, going the other way would be a problem because the value of the *long* might exceed the capability of the *int*, so an explicit cast is required:

```
i = (int)l;
```

Although there's obviously no problem in this particular example, in the general case the value of the *long* would be truncated to fit in an *int*. Similarly, you need an explicit cast to assign a *long* to a *short*, or an *int* to a *short*.

C# also supports unsigned integral data types, but the word "unsigned" is not a keyword as it is in C and C++. The three unsigned integral data types are named *ushort*, *uint*, and *ulong*. The *ushort* ranges in value from 0 through 65,535; the *uint* from 0 through 4,294,967,295, and the *ulong* from 0 through 18,446,744,073,709,551,615.

You can assign a *ushort* to a *uint* or a *ulong* without an explicit cast, or a *uint* to a *ulong* without an explicit cast because nothing bad can ever happen. You can also assign a *ushort* to an *int*, or a *uint* to a *long* without an explicit cast. Again, the recipient can accommodate all possible values of the source. But explicit casts are required anywhere the value may not survive. For example, setting an *int* to a *uint*, or a *uint* to an *int* always requires an explicit cast.

Numeric literals are generally assumed to be of type *int*, except if the number is too big for an *int*, in which case, the C# compiler assumes (progressively) that it's actually a *uint*, *long*, or *ulong*. You can be more specific by applying a suffix of *U* or *u* (for unsigned) or *L* or *l* (for *long*) or a combination of *U* or *u* and *L* or *l*.

You can represent hexadecimal numbers by proceeding the digits with a 0x or 0X:

```
int hex = 0x4AbC;
```

You can't express octal or binary literals in C#.

The *long* and *ulong* types are 64-bits wide, the *int* and *uint* types are 32-bit wide, and the *short* and *ushort* types are 16-bit wide. C# also supports two 8-bit integral types, but the naming convention is reversed. The *byte* is an unsigned 8-bit type capable of storing values from 0 through 255. The *sbyte* is the signed 8-bit type capable of storing values from -128 to 127.

Implicit casts are allowed for converting *byte* to *short*, *ushort*, *int*, *uint*, *long*, and *ulong*; and from *sbyte* to *short*, *int*, and *long*.

A program can also declare constants:

```
const int a = 17;
```

A data type must follow the *const* keyword, and the constant must be initialized in the declaration statement. You can declare a *const* either locally to a method or as a field. If you declare a *const* as a field, do not use the *static* keyword with *const*. Constants are implicitly static, that is, they don't vary by instance. The *Math.PI* field is a *const*.

Constants are sometimes treated as variables that cannot be varied, but they are actually quite different from variables. Constants do not occupy storage during runtime. They are always evaluated at compile time and substituted into code. For example, if the constant *a* has been defined as shown above, the statement

```
int b = a + 18;
```

is equivalent to:

```
int b = 17 + 18;
```

The initialized value of a *const* must be available at compile time.

If a program divides an integer by zero, a *DivideByZeroException* will be raised. If a program contains an expression that divides an integer by a literal 0 or a *const* with a value of 0, the C# compiler is alert enough to flag that as a compile error.

By default, no exception is raised when integer overflow or underflow occurs. For example:

```
int i = 50000;
Console.WriteLine(i * i);
```

That statement will print the value -1794967296 (and if you don't know why it's negative, you can refer to Chapter 13 of my book *Code: The Hidden Language of Computer Hardware and Software* or Chapter 8 of *Programming in the Key of C#*).

If you'd prefer that your integer arithmetic is checked for overflow and underflow—perhaps just during program development—you can set a compiler switch. When compiling on the command line, use the switch:

```
/checked+
```

In Visual Studio, bring up Project Properties, select the Build tab, click the Advanced button, and check the checkbox labeled "Check for arithmetic overflow/underflow". Now any overflow or underflow raises an *OverflowException*.

You can also perform overflow/underflow checking on individual expressions or groups of statement using the *checked* keyword. For example, this statement performs checking on just one expression:

```
B = checked(5 * A);
```

If you have turned on overflow/underflow checking with the compiler switch, you can turn it off for individual expressions with the *unchecked*

keyword. Perhaps you need to keep a particular expression unchecked because you use overflow deliberately here:

```
B = unchecked(5 * A);
```

You can also perform overflow/underflow checking for a particular block of statements:

```
checked
{
    A += 1000000;
    B = B * A;
}
```

Similarly, you can turn off checking for a particular block of statements.

Just keep in mind that the most efficient code is that in which overflow and underflow are not checked during runtime.

Integers and the .NET Framework

The *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, and *ulong* keywords are aliases for structures in the .NET Framework class library. For example, if you look in the documentation of the *System* namespace, you'll find a structure named *Int32*. This is the structure that corresponds to the unsigned 32-bit *int*. You can alternatively define an *int* named *i* like this:

```
System.Int32 i;
```

Or, if you have a *using* directive for the *System* namespace, you can use this:

```
Int32 i;
```

Anywhere you use *int*, you can alternatively use *System.Int32* or (if the *using* directive exists) *Int32*. Jeff Richter actually recommends that C# programmers use the .NET structure names. (See *CLR via C#*, page 119.) As you'll discover, you sometimes need to refer to types by the structure name, and it helps if you've become accustomed to those names to begin with.

The following table shows the eight integral data types and their corresponding structures in the .NET Framework:

C# Alias	.NET Structure	Description	CLS Compliant?
sbyte	System.SByte	Signed 8-bit integer	No
byte	System.Byte	Unsigned 8-bit integer	Yes
short	System.Int16	Signed 16-bit integer	Yes
ushort	System.UInt16	Unsigned 16-bit integer	No
int	System.Int32	Signed 32-bit integer	Yes
uint	System.UInt32	Unsigned 32-bit integer	No

long	System.Int64	Signed 64-bit integer	Yes
ulong	System.UInt64	Unsigned 64-bit integer	No

In the last column I've indicated if the particular data type is compliant with the Common Language Specification (CLS). The CLS specifies a minimum standard for programming languages used in conjunction with .NET. The CLS does not require programming languages to implement signed 8-bit integers, or unsigned 16, 32, or 64 bit integers. What this means is that a particular programming language can be CLS compliant without implementing these data types.

There is no reason why you shouldn't use these data types in your C# programs. *However*, if you are writing code for a dynamic link library, then you should certainly avoid these data types to define public fields, or as parameters or return values from public methods, or as types of public properties. Methods in the DLL should not have *uint* arguments or return *uint* values because that would prohibit the method from being used by a program written in a language that does not implement the *UInt32* data type.

Does defining these data types as structures in the .NET Framework offer any benefits?

Yes. The *ToString* method defined in the *Int32* structure is particularly tailored for formatting integers, for example. Also, each of the structures for the numeric data types defines public *const* fields named *MinValue* and *MaxValue*. The *Int16* structure defines:

```
public const short MinValue = -32768;
public const short MaxValue = 32767;
```

You can refer to these fields in your programs by prefacing them with the *Int16* structure name:

```
Console.WriteLine(Int16.MinValue);
```

If you didn't have a *using* directive for the *System* namespace, this statement would have to be:

```
System.Console.WriteLine(System.Int16.MinValue);
```

You can use the fully-qualified name even if you have a *using* directive for *System* namespace. You can also preface the fields with the *short* alias:

```
Console.WriteLine(short.MinValue);
```

Boolean Data Types

C# defines a *bool* data type, which is an alias for the *System.Boolean* structure. A *bool* variable can have one of two possible values, which are equivalent to the C# keywords *true* and *false*.

One of the naming conventions that has emerged in the .NET Framework is that Boolean methods and properties often begin with the word *Is*. Local fields or variables might begin with a lower-case *is*, like so:

```
bool isGreatDay = true;
```

As you'll discover in the next chapter, several common C# operators return *bool* values. These are the equality operators (== and !=) and the relational operators (<, >, <=, and >=). You can use the logical operators (&, |, and ^) and the conditional operators (&& and ||) with *bool* values.

The Boolean structure defines a *ToString* method that displays the words "True" or "False", and a *Parse* method that implements a case-insensitive conversion of strings of "true" or "false" to *bool* values.

Character and String Data Type

Like C and C++, C# defines a data type named *char*, but strictly speaking, *char* is not an integral data type in C#. Although you can easily convert between *char* and integral data types, there is no such thing as a signed *char* or an unsigned *char*.

The *char* data type in C# is an alias for the *System.Char* structure. A program can specify a single literal character using single quotation marks:

```
char ch = 'A';
```

Converting a *char* variable to an integer requires casting:

```
int i = (char) ch;
```

In C# the *char* stores one Unicode character, which requires 16 bits. Character variables thus have numeric values from 0x0000 to 0xFFFF.

As with string literals, the backslash is an escape character. The following declaration statement initializes the character variable to the Greek capital omega (Ω):

```
char omega = '\x03A9';
```

Or, you can cast an integer to a *char*:

```
char omega = (char) 0x03A9;
```

As you know, in C or C++, you can use functions declared in the ctype.h header file to determine whether a particular character is a letter, number, control character, or whatever. In C#, you use static methods defined in the *Char* structure: *IsControl*, *IsSeparator*, *IsWhiteSpace*, *IsPunctuation*, *IsSymbol*, *IsDigit*, *IsNumber*, *IsLetter*, *IsUpper*, *IsLower*, *IsLetterOrDigit*, *IsSurrogate*. These methods come in two versions, one which has a single parameter of type *char*, and the other which accepts a *string* and an index. The call

```
Char.IsControl(str[index]);
```

is equivalent to:

```
Char.IsControl(str, index);
```

All these method return *bool* values.

The *string* data type is an alias for the *System.String* class. A string is a consecutive collection of *char* objects. The *Length* property defined by the *String* class indicates the number of characters in a string. An indexer obtains individual characters. The *String* class defines a multitude of methods for working with strings, which you can explore on your own.

Because strings are immutable, it is easy to write code that looks very innocent but which is very inefficient. See Chapter 26 for problems and solutions.

Floating Point

Like C and C++, C# supports floating point data types named *float* and *double* that correspond to the single-precision and double-precision specifications defined in ANSI/IEEI Std 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*.

```
double num1 = 576.34;
float num2 = 34.89f;
```

By default, the C# compiler will assume that any numeric literal with a decimal point is a *double*. If you want it instead to be a *float*, you must use an *F* or *f* suffix, as is shown in the second declaration statement. You can use a *D* or *d* suffix to explicitly indicate a *double*. At least one digit must be specified after the decimal point. You can use a capital or lower-case E to indicate an exponent:

```
double num1 = 5.7634E2;
float num2 = 3489.0e-2f;
```

The *float* type is an alias for the *System.Single* structure, and the *double* type is an alias for the *System.Double* structure. (Notice that the C# data type is *float* but the structure is named *Single*.)

A *float* value consists of a 24-bit signed mantissa and an 8-bit signed exponent. The precision is approximately seven decimal digits. Values range from

$$-3.402823 \times 10^{38}$$

to:

$$3.402823 \times 10^{38}$$

The smallest possible float value greater than 0 is:

$$1.401298 \times 10^{-45}$$

You can obtain these three values from the *MinValue*, *MaxValue*, and *Epsilon* constant fields defined in the *Single* structure.

A *double* value consists of a 53-bit signed mantissa and an 11-bit signed exponent. The precision is approximately 15 to 16 decimal digits. Values range from

$-1.79769313486232 \times 10^{308}$

to:

$1.79769313486232 \times 10^{308}$

The smallest possible *double* value greater than 0 is:

$4.9465645841247 \times 10^{-324}$

The *MinValue*, *MaxValue*, and *Epsilon* fields are also defined in the *Double* structure.

Implicit casting is allowed from *float* to *double*, and from any integral type to *float* or *double*. Explicit casting is required from *double* to *float*, or from *double* or *float* to any integral type. In arithmetical expressions that mix *float* and integral types, integers are converted to *float* for the calculation. If the expressions involve *double*, any integers or *float* values will be converted to *double*.

Here's some code that divides a floating-point number by zero:

```
double d1 = 1;
double d2 = 0;
double d3 = d1 / d2;
```

If these were integers, a *DivideByZeroException* would be raised. But these are IEEE floating-point numbers. An exception is not raised. Floating-point operations *never* raise exceptions in C#. Instead, in this case *d3* takes on a special value. If you use *Console.WriteLine* to display *d3*, it will display the word

```
Infinity
```

If you change the initialization of *d1* to -1, *Console.WriteLine* will display:

```
-Infinity
```

In the IEEE standard, positive infinity and negative infinity are legitimate values of floating-point numbers. You can even perform arithmetic on infinite values. For example, the expression

```
1 / d3
```

equals 0.

If you change the initialization of *d1* in the preceding code to 0, then *d3* will equal a value known as *Not a Number*, which is abbreviated as *NaN* and pronounced "nan." Here's how *Console.WriteLine* displays a NaN:

```
NaN
```

You can create a NaN by adding positive infinity to negative infinity or by a number of other calculations.

Both the *Single* and *Double* structures have static methods named *IsInfinity*, *IsPositiveInfinity*, *IsNegativeInfinity*, and *IsNaN* to determine whether a particular *float* or *double* value is infinity or NaN. These methods require a floating-point argument and return a *bool*. For example,

```
Double.IsInfinity(d)
```

returns *true* if *d* is either positive infinity or negative infinity.

The *Single* and *Double* structures also have constant fields named *PositiveInfinity*, *NegativeInfinity*, and *NaN* to represent these values. These values correspond to specific bit patterns in the IEEE standard. However, these bit patterns are not unique, so it is not recommended that you use these fields for comparison operations. For example, even if *d* is a NaN, the operation

```
d == Double.NaN
```

will return *false* if the bit pattern of *d* does not correspond exactly with that of *Double.NaN*. Use the static methods for determining the status of particular numbers:

```
Double.IsNaN(d)
```

Much confusion surrounds the floating-point remainder operation. The C# remainder or modulus operator (%) is defined for all numeric types. (In C, the modulus operator is not defined for *float* and *double*, and the *fmod* function must be used instead.) Here's a C# statement using *double* numbers with the remainder operator:

```
result = dividend % divisor;
```

The sign of *result* is the same as the sign of *dividend*, and *result* can be calculated with the formula

```
result = dividend – n * divisor
```

where *n* is the largest possible integer less than or equal to *divided* / *divisor*. For example, the expression

```
4.5 % 1.25
```

equals 0.75. (The expression 4.5 / 1.25 equals 3.6, so *n* equals 3. The quantity 4.5 minus (3 times 1.25) equals 0.75.

The IEEE standard defines a remainder a little differently, where *n* is the integer *closest* to *dividend* / *divisor*. You can obtain a remainder in accordance with the IEEE standard using the static *Math.IEEERemainder* method. The expression

```
Math.IEEERemainder(4.5, 1.25)
```

equals -0.5. That's because 4.5 / 1.25 equals 3.6, and the closest integer to 3.6 is 4. When *n* equals 4, the quantity 4.5 minus (4 times 1.25) equals -0.5.

Decimal Data Type

C# also defines a *decimal* data type that offers about 28 decimal digits of precision. The *decimal* is useful for storing and calculating numbers with a fixed number of decimal points, such as money and interest rates.

In my book *Programming in the Key of C#*, I deliberately covered *decimal* before floating point. I think it's important for new programmers to use *decimal* for most applications involving non-integral data types, particularly when the calculations involve money. The C and C++ languages were not developed in a tradition that valued adequate tools for financial applications. The *decimal* is an attempt to correct that historical deficiency.

The *decimal* data type is an alias for the *System.Decimal* structure. For *decimal* more than any of the other numeric data types, that structure is of vital importance because *decimal* is not supported by Common Intermediate Language.

What does that mean? CIL supports integral types and the two floating point types directly, but not the *decimal* type. When you write C# code to multiply two *double* values, for example, the C# compiler generates intermediate language to push the two values on the stack, followed by a CIL *mul* instruction. At runtime, this intermediate language is converted into machine code that uses the math coprocessor.

But there is no CIL *mul* instruction for *decimal*. Instead, *decimal* is supported almost entirely through the *System.Decimal* structure. When you write C# code to multiply two *decimal* numbers, the multiplication is actually performed by the *op_Multiply* method defined by the *Decimal* structure. (This method name refers to an overload of the multiplication operator defined by the *Decimal* class. I discuss operator overloading in Chapter 20.)

Although the CIL does not directly support the *decimal* data type, it is nonetheless part of the Common Language Specification. A .NET language must support the *Decimal* structure, but this support can be fairly minimal. In C#, it amounts to little more than recognizing decimal literals, which are denoted with an *m* or *M* suffix:

```
decimal m = 55.23m;
```

Leaving out the *m* will result in a compile error. The literal will be assumed to be a *double*, and there are no implicit conversions between the floating point types and *decimal*.

Calculations involving *decimal* cannot be controlled using the *checked* and *unchecked* keywords or the related compiler switch. Calculations that result in overflow or underflow always raise an *OverflowException*.

The *decimal* type uses 16 bytes (128 bits) to store each value, which is twice as many bits as the *double*. The 128 bits break down into a 96-bit integral part, a 1-bit sign, and a scaling factor that can range from 0 through 28. (Twenty-six bits are unused.) Mathematically, the scaling factor is a negative power of 10 and indicates the number of decimal places in the number.

Don't confuse the decimal type with a binary-coded decimal (BCD) type found in some programming languages. A BCD type stores each decimal digit using 4 bits. The *decimal* type stores the entire number in binary. For example, a decimal equal to 12.34 is stored as the integer 0x4D2 (or decimal 1,234) with a scaling factor of 2, which denotes a multiplication by 10^{-2}. A BCD encoding of 12.34 would store the number as 0x1234.

As long as a number has 28 significant decimal digits (or fewer) and 28 decimal places (or fewer), the *decimal* data type stores the number exactly. This is not true with floating point! If you define a *float* equal to 12.34, it's essentially stored as the value 0xC570A4 (or 12,939,428) divided by 0x100000 (or 1,048,576), which is only *approximately* 12.34. Even if you define a *double* equal to 12.34, it's stored as the value 0x18AE147AE147AE (or 6,946,802,425,218,990) divided by 0x2000000000000 (or 562,949,953,421,312), which again only approximately equals 12.34.

And that's why you should use *decimal* when you're performing calculations where you don't want pennies to mysteriously crop up and disappear. The floating-point data types are great for scientific and engineering applications but often undesirable for financial ones.

Implicit conversions are allowed from all the integer types to *decimal*, and it's easy to see why. Explicit casts are required going the other way, and a runtime *OverflowException* will result if the decimal number is too large to fit in the destination integral type. Explicit casts are required from either floating-point type to *decimal* because the exponents of floating-point allow values unrepresentable by *decimal*. Explicit casts are also required for conversions from *decimal* to either floating-point type because *decimal* allows greater precision.

The *decimal* type also includes constructors that accept an integral or floating-point data type. These are mostly for languages that don't explicitly support *decimal*.

If you want to explore *decimal* a bit, you can make use of another constructor that lets you put together a *decimal* from its constituent parts:

```
decimal m = new decimal(low, middle, high, isNegative, scale);
```

The first three arguments are defined as *int* but are treated as if they were unsigned integers. (If they were defined as unsigned integers, this

constructor wouldn't be CLS compliant.) The three 32-bit values become the 96-bit integral part of the *decimal*. The *isNegative* parameter is a *bool* indicating if the number is negative. The *scale* argument can range from 0 to 28 to indicate the number of decimal points.

The expression

```
new decimal(1234567, 0, 0, false, 5)
```

creates the *decimal* number 12.34567. The largest positive *decimal* number is

```
new decimal(-1, -1, -1, false, 0)
```

or 79,228,162,514,264,337,593,543,950,335, which you can also obtain from the *Decimal.MaxValue* field. The smallest decimal number closest to 0 is

```
new decimal(1, 0, 0, false, 28)
```

or 0.0000000000000000000000000001 or 1×10^{-28}. If you divide this number by 2 in a C# program, the result is 0.

It is also possible to obtain the bits used to store a *decimal* value using the static *GetBits* method. This method returns an array of four integers. To get the four *int* values that make up a *decimal*, you need to declare an array of type *int* and call *Decimal.GetBits* with a *decimal* argument:

```
int[] A = Decimal.GetBits(m);
```

The first, second, and third elements of the array (that is, *A[0]*, *A[1]*, and *A[2]*) are the low, medium, and high components of the 96-bit unsigned integer.

The fourth element contains the sign and the scaling factor. Bits 0 through 15 are 0; bits 16 through 23 contains a scaling factor between 0 and 28; bits 24 through 30 are 0; and bit 31 is 0 for positive and 1 for negative. In other words, if *A[3]* is negative, the decimal number is negative. The scaling factor is:

```
(A[3] >> 16) & 0xFF
```

Almost everyone who has worked extensively with floating-point can recall incidences in which a calculated number that should have been 4.55 (for example) is often stored as 4.549999 or 4.550001. The *decimal* representation is much better behaved. Suppose *m1* is defined like so:

```
decimal m1 = 12.34m;
```

Internally *m1* has an integer part of 1234 and a scaling factor of 2. Suppose *m2* is defined like this:

```
decimal m2 = 56.789m;
```

The integer part is 56789 and the scaling factor is 3. Now add these two numbers:

```
decimal m3 = m1 + m2;
```

Conceptually, the integer part of *m1* is multiplied by 10 (to get 12340), and the scaling factor is set to 3. Now the integer parts can be added directly: 12340 plus 56789 equals 69129 with a scaling factor of 3. The actual number is 69.129. Everything is exact.

Now multiply the two numbers:

```
decimal m4 = m1 * m2;
```

The two integral parts are multiplied (1234 times 56789 equals 70,776,626), and the scaling factors are added (2 plus 3 equals 5). The actual numeric result is 700.77626. Again, the calculation is exact.

When dividing… well, division is messy no matter how you do it. But for the most part, when using *decimal*, you can have much greater confidence in the precision and accuracy of your results.

Math Class

The *Math* class in the *System* namespace consists entirely of a collection of static methods and the two constant fields. The two fields of type *double* are named *PI* and *E*. *Math.PI* is the ratio of the circumference of a circle to its diameter, or 3.14159265358979. *Math.E* is the limit of

$$\left(1+\frac{1}{n}\right)^n$$

as *n* approaches infinity, or 2.71828182845905.

Most of the methods in the *Math* class are defined only for *double* values. However, some methods are defined for integer and *decimal* values as well. The *Max* and *Min* methods both accept two arguments of the same numeric type and return the maximum or minimum, respectively.

The *Abs* and *Sign* methods are defined for floating-point types, *decimal* types, and signed integer types. The *Abs* method returns the absolute value of the argument. The *Sign* method returns an *int*: 1 if the argument is positive, –1 if the argument is negative, and 0 if the argument is 0.

The *Abs* method is the only method of the *Math* class that can raise an exception, and then only for integral arguments, and only for one particular value of each integral type, namely the value stored in the *MinValue* field. The method call:

```
short s = Math.Abs(Int16.MinValue);
```

raises an *OverflowException* because *Int16.MinValue* is –32,768 and 32,768 can't be represented by a *short*.

The *BigMul* and *DivRem* methods were introduced in .NET 2.0 and are defined for integers. *BigMul* accepts two *int* arguments and returns a *long*:

```
long l = Math.BigMul(i1, i1);
```

You can get the same result if you first cast one of the arguments to a *long*:

```
long l = (long)i1 * i2;
```

The *DivRem* method is defined for both *int* and *long*. In both cases, it has three arguments and one return value of the same type. The return value is the integer division of the first two arguments. The third argument receives the remainder. For example, if *a*, *b*, *c*, and *d* are all defined as *int* (or all defined as *long*) then you call *DivRem* like this:

```
c = Math.DivRem(a, b, out d);
```

The *a* and *b* variables must be initialized before calling the method, but *d* does not. The *c* variable receives the result of the integer division, and *d* get the remainder. Notice the *out* keyword that indicates that *d* is being passed by reference to the method and then set from the method. I'll have more to say about *out* in Chapter 11. The *DivRem* method is functionally identical to the code:

```
c = a / b;
d = a % b;
```

The *Floor* and *Ceiling* methods are defined for *double* arguments only. *Floor* returns the largest whole number less than or equal to the argument. *Ceiling* returns the smallest whole number greater than or equal to the argument. The call

```
Math.Floor(3.5)
```

returns 3, and

```
Math.Ceiling(3.5)
```

returns 4. The same rules apply to negative numbers. The call

```
Math.Floor(-3.5)
```

returns –4, and

```
Math.Ceiling(-3.5)
```

returns –3.

The *Floor* method returns the nearest whole number in the direction of negative infinity, and that's why it's sometimes also known as "rounding toward negative infinity"; likewise, *Ceiling* is sometimes known as "rounding toward positive infinity."

It's also possible to round toward 0, which is to obtain the nearest whole number closest to 0. You round toward 0 by casting to an integer. The expression

```
(int) 3.5
```

returns 3, and

```
(int) -3.5
```

returns –3. Rounding toward 0 is commonly known as "truncation."

The *Round* method is defined for both *double* and *decimal*. The version with a single argument returns the whole number nearest to the argument. If the argument to *Round* is midway between two whole numbers, the return value is the nearest even number. For example, the call

```
Math.Round(4.5)
```

returns 4, and

```
Math.Round(5.5)
```

returns 6. Although the return value is always a whole number, the type of the return value is the same as the type of the argument (*double* or *decimal*).

If you prefer the convention where numbers ending in .5 always round up, add 0.5 to the number you wish to round and truncate. Or, you can use one of the new overloads of *Round* introduced in .NET 2.0 that have an enumeration argument:

```
Math.Round(4.5, MidpointRounding.ToEven)
```

returns 4 but

```
Math.Round(4.5, MidpointRounding.AwayFromZero)
```

returns 5. These enumeration values only affect the result when the number is midway between two integers.

You can optionally supply an integer to *Round* that indicates the number of decimal places in the return value. For example,

```
Math.Round(5.285, 2)
```

returns 5.28. And in .NET 2.0 and later you can supply an integer and an enumeration value. The call

```
Math.Round(5.285, 2, MidpointRounding.AwayFromZero)
```

returns 5.29.

Three methods of the *Math* class involve powers. The first is *Pow*:

```
Math.Pow(base, power)
```

The method returns:

$$base^{power}$$

The method returns *NaN*, *NegativeInfinity*, or *PositiveInfinity* in some cases. See the documentation for details.

The expression

```
Math.Exp(power)
```

is equivalent to:

```
Math.Pow(Math.E, power)
```

and the expression

```
Math.Sqrt(value)
```

is equivalent to:

```
Math.Pow(value, 0.5)
```

The *Math* class has three methods that calculate logarithms. The expression

```
Math.Log10(value)
```

is equivalent to

```
Math.Log(value, 10)
```

and

```
Math.Log(value)
```

is equivalent to

```
Math.Log(value, Math.E);
```

The three basic trigonometric functions *Math.Sin*, *Math.Cos*, and *Math.Tan* require that angles be specified in radians. There are 2π radians in 360 degrees. If *angle* is in degrees, call *Math.Sin* like this:

```
Math.Sin(Math.PI * angle / 180)
```

The *Math.Sin* and *Math.Cos* methods return values ranging from –1 to 1. In theory, the *Math.Tan* method should return infinity at π/2 (90 degrees) and 3π/2 (270 degrees) but it returns very large values instead.

The inverse trigonometric functions return angles in radians. The following table shows the return values for proper ranges of arguments:

Method	**Argument**	**Return Value**
Math.Asin(value)	–1 through 1	–π/2 through π/2
Math.Acos(value)	–1 through 1	π through 0
Math.Atan(value)	–∞ through ∞	–π/2 through π/2
Math.Atan(y, x)	–∞ through ∞	–π through π

To convert the return value to degrees, multiply by 180 and divide by π. The *Asin* and *Acos* methods return NaN if the argument is not in the proper range. The *Atan2* method uses the sign of the two arguments to determine the quandrant of the angle:

y Argument	x Argument	Return Value
Positive	Positive	0 through π/2
Positive	Negative	π/2 through π
Negative	Negative	–π through –π/2
Negative	Positive	–π/2 through 0

Less commonly used are the hyperbolic trigonometric functions *Math.Sinh*, *Math.Cosh*, and *Math.Tanh*. Angle arguments are in hyperbolic radians.

Chapter 7. Operators and Expressions

The following table is so important that you might want to print out an extra copy of this page:

Operator Precedence and Associativity		
Category	**Operators**	**Associativity**
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	Left to right
Unary	+ - ! ~ ++x --x (T)x	Left to right
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< > <= >= is as	Left to right
Equality	== !=	Left to right
Logical AND	&	Left to right
Logical XOR	^	Left to right
Logical OR	|	Left to right
Conditional AND	&&	Left to right
Conditional OR	||	Left to right
Conditional	?:	Right to left
Assignment	= *= /= %= += -= <<= >>= &= ^= |= ??	Right to left

Veteran C programmers will recognize this as a variation of a table on page 49 in the first edition of Brian Kernighan and Dennis Ritchie's *The C Programming Language* (Prentice-Hall, 1978). The C# version of the

table appears in §7.2.1 of the *C# Language Specification*. The comma operator in C and C++ is not supported in C#.

When working with C#, it is advantageous to think in terms of operators and expressions rather than statements. Operators are generally symbols or small words that cause changes to variables, or which use the values of a variable or multiple variables to produce a single result. An expression involves one or more operators. For example,

```
A + B
```

is an expression. It's not a statement, even if you put a semicolon after it. (C# doesn't allow the addition operator to be used by itself as a statement because the statement doesn't do anything.) The table shown above indicates how the C# compiler evaluates complex expressions.

For example, if you mix a multiplicative operator and an additive operator in the same statement, the multiplicative operator will be evaluated first. In the statement

```
C = A + B * 5;
```

the multiplication occurs first and then the addition.

If two operators have the same precedence, then the order is governed by the associativity. For example:

```
C = A / B * 5;
```

Both are multiplicative operators. The division occurs first, followed by the multiplication. All associativity is left to right except for the conditional and assignment operators.

If the default precedence and associativity is not what you want, you can use parentheses to change it:

```
C = (A + B) * 5;
```

Now the addition occurs first. The pair of parentheses is considered an operator, and it is the first operator listed in the Primary operators row.

Approaching C# in terms of expressions and operators is helpful in understanding what's going on here:

```
Console.WriteLine(C = A + B);
```

Some languages won't let you do that because assignment is not considered an operator like the plus sign. (Visual Basic .NET, for example, will interpret the equal sign as an equality operator.) In C#, the addition is evaluated first. As you can see in the chart, Additive operators have higher precedence than Assignment operators. The result of the addition operation then becomes the right hand side of the assignment operation. The equal sign sets the variable on the left equal to the value on the right. Also—and this is crucial in the example—the assignment operation

has a result, and that result is the new value of the variable on the left. That result is the value passed to the *Console.WriteLine* method.

Primary Operators

You've already seen many of the operators in the first row of the table:

The **(x)** operator symbolizes parentheses, which can surround an expression to change the order of precedence or associativity.

The **x.y** operator symbolizes the period, which you've seen separate namespace names, class and structure names, and method, field, and property names.

The **f(x)** operator symbolizes method calls (where 'f' stands for "function," of course).

The **a[x]** operator symbolizes array access and indexing, such as with strings.

The **x++** and **x--** operators symbolize the post-increment and post-decrement operators. The variable is incremented or decremented, but only after the variable is used in an expression. For example,

```
int A = 55;
Console.WriteLine(A++);
Console.WriteLine(A);
```

The first *Console.WriteLine* displays 55 because that's the value before the incrementation. The second *Console.WriteLine* displays 56.

You've also seen the **new** operator, as well as **checked** and **unchecked**. I'll discuss **typeof** in Chapter 19. I won't be discussing **sizeof** in this book.

Unary Operators

These operators are called "unary" because they have only one argument, which always appears to the right of the operator. The **+** operator is defined for all numeric types but normally does nothing. For example, the expression

```
+A
```

is just *A*. However, the operator could be overridden by a class or structure.

The result of the **–** operator is the negative of a number. (The operator doesn't change the variable itself.)

The **!** operator is logical negation and is defined only for *bool*. When applied to a *true* value, the result is *false*, and when applied to *false*, the result is *true*.

The **~** is the bitwise complement operator and is defined only for *int*, *uint*, *long*, and *ulong*. The operator results in an integer of the same type in which all the bits have been flipped from 0 to 1, and from 1 to 0.

The pre-increment and post-increment operators are symbolized by **++x** and **--x**. The variable is incremented or decremented, and that's also the result of the operation.

The operator **(T)x** symbolizes casting. C# is much stricter than C with regard to casting. If you need to convert from one data type to another beyond what C# allows, the *Convert* class (defined in the *System* namespace) provides many methods that probably do what you want.

Multiplicative and Arithmetic Operators

The multiplicative and arithmetic operators are defined for all numeric data types. The addition operator is also defined for *string*. Unlike C and C++, the remainder operator is defined for *float* and *double*, as I discussed in the previous chapter.

Shift Operators

The shift operators shift an integer (on the left of the operator) by a specified number of bits (the operand on the right). The **<<** operator is a left shift; the **>>** operator is a right shift.

The operand on the left must be an *int*, *uint*, *long*, or *ulong*. The operand on the right must be an *int*. When shifting an *int* or *uint*, only the bottom 5 bits of the operand on the right are used; when shifting a *long* or *ulong*, only the bottom 6 bits of the right operand are used.

When an *int* or *long* is shifted right, the shift is arithmetic. That is, the vacated high-order bits are set to the sign bit.

Relational Operators

The first four relational operators, **<**, **>**, **<=**, and **>=** are defined for all numeric types and return a *bool* value.

The two type-testing operators *is* and *as* will be discussed in Chapter 19.

Equality Operators

The two equality operators **==** and **!=** are defined for all numeric types. They return a *bool* value.

There are some special considerations for the equality operators, which I discuss in Chapter 16. For now, you might be pleased to know that the equality operators work with *string* objects. Two strings are considered equal if they have the same length and corresponding characters. The comparison is case-sensitive. Other types of comparisons are possible with methods defined by the *String* class.

Logical Operators and Conditional Operators

I want to discuss the two AND and OR operators in some detail because they can cause some confusion among C and C++ programmers.

The **&**, **^**, and **|** operators are termed the *logical* AND, XOR, and OR. (In C, these are called the *bitwise* operators.) In C# these operators are defined for both integral data types and *bool*. For integral data types, they function as bitwise operators, the same as in C. For example, the expression:

```
0x03 | 0x05
```

evaluates as 0x07.

For *bool* data types, the logical operators evaluate to a *bool* result. The result of the logical AND operator is *true* only if both operands are *true*. The result of the logical XOR is *true* only if one operand is *true* and the other is *false*. The result of the logical OR is *true* if either of the operands is *true*.

In C, the **&&** and **||** operators are known as *logical* operators. In C#, they're termed *conditional* AND and OR, and they are defined only for *bool* data types.

C programmers are accustomed to using the && and || operators in expressions like this:

```
A != 0 && B > 5
```

C programmers also know that if the first expression evaluates as *false* (that is, if A equal 0), then the second expression isn't evaluated. It's important to know this because the second expression could involve an assignment or a function call. Similarly, when you use the || operator, the second expression isn't evaluated if the first expression is *true*.

In C# you use the && and || operators in the same way you use them in C. These operators are called the *conditional* AND and OR because the second operand is evaluated only if necessary.

In C#, you can use the & and | operators in the same way as && and ||, as in this example:

```
A != 0 & B > 5
```

When you use the & and | operators in this way in C#, both expressions are evaluated regardless of the outcome of the first expression.

The statement using & rather than && is certainly legal in C, and it works the same way as in C#, but most C programmers would probably write such a statement only in error. The statement looks wrong to many C programmers because they've trained themselves to treat the & as the bitwise AND and the && as the logical AND. But in C the result of a relational or logical expression is an *int* that has a value of 1 if the

expression is *true* and 0 otherwise. That's why the bitwise AND operation works here.

A C programmer might make the original statement involving the && operator a little more concise by writing it like so:

```
A && B >= 5
```

This works fine in C because C treats any nonzero expression as being *true*. In C#, however, this statement is illegal because the && operator is defined only for *bool* data types.

Where the C programmer gets into big trouble is using the bitwise AND operator in the shortened form of the expression:

```
A & B >= 5
```

If *B* equals 7, then the expression on the right is evalulated as the value 1. If *A* is equal to 1 or 3 or any odd number, then the bitwise AND operation yields 1 and the total expression evaluates as *true*. If A is 0 or 2 or any even number, then the bitwise AND operation yields 0 and the total expression evalulates as *false*. It's likely that none of these results are what the programmer intended, and this is precisely why the C programmer has such a distressed reaction to seeing bitwise AND and OR operators in logical expressions. (In C# this statement is illegal because integers and *bool* values can't be mixed in the logical AND, XOR, and OR statements.)

Suppose you have an integer variable named *year* and you wish to calculate a *bool* named *isLeapYear*. Leap years are generally those years divisible by 4, except that years divisible by 100 are not leap years, except if the year is also divisible by 400. You could do it this way:

```
bool isLeapYear = year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
```

The first expression to be evaluated is:

```
year % 4 == 0
```

That expression will be *true* only if year is equally divisible by 4. The expression

```
year & 3 == 0
```

would also do the job. The single ampersand is a bitwise AND in this expression, and the result is 0 only if *year* is equally divisible by 4.

Either way, three-quarters of all years are eliminated immediately. Because the next operator is a conditional AND, the rest of the expression doesn't even get evaluated for three-quarters of all years. The remainder of the expression is enclosed in parentheses:

```
(year % 100 != 0 || year % 400 == 0)
```

This expression is evaluated only if *year* is divisible by 4. It is intended to eliminate those years divisible by 100 but not divisible by 400. This

expression is *true* if *year* is divisible by 400, or if *year* is not divisible by 100.

Conditional Operator

The conditional operator, which is symbolized by **?:** in the table, is the only ternary operator, which means it has three operands:

```
A ? B : C
```

The expression *A* must resolve to a *bool*. The result of the operation is *B* if *A* is *true*, and *C* if *A* is *false*.

The use of this operator is customarily restricted to special occasions, such as this code that appends an 's' to some text only if the value being displayed is not equal to 1:

```
Console.WriteLine("Please deposit {0} dollar{1}.",
                  dollars, dollars == 1 ? '' : 's');
```

Assignment Operators

Assignment is a binary operation, and the result of an assignment expression is the value being assigned. Assignment is evaluated right to left, making possible statements like this one:

```
A = B = 3;
```

B is assigned the value 3, and that is also the result of the assignment expression. That result is then assigned to *A*.

Like C and C++, C# also supports the popular compound assignment operators for addition, subtraction, multiply, divide, remainder, shift, and AND, OR, and XOR in both their bitwise and logical interpretations.

The final operator listed in the table is **??**, called the *null coalescing operator*, which I'll describe in Chapter 27.

Chapter 8. Selection and Iteration

C# supports the same selection, iteration, and flow control statements used in C and C++, but generally with some restrictions. These restrictions are not severe and generally are intended to help you avoid common coding bugs. In this chapter I'll discuss statements built around the *if*, *else*, *switch*, *case*, *default*, *do*, *while*, *for*, *foreach*, *in*, *break*, *continue*, and *goto* keywords.

Selection Statements

The basic selection statement involves the *if* and *else* keywords. The keyword *if* must be followed by a Boolean expression in parentheses. The statement that follows is executed if the Boolean expression resolves to *true*:

```
if (a < 5)
    b += 27;
```

The requirement that the parentheses contain a Boolean expression eliminates a whole class of common C bugs. Almost every C and C++ programmer has committed the common pitfall of mistakenly using an assignment as the test expression when a comparison was intended:

```
if (a = 5)
```

The C# compiler flags this statement as an error, and you'll probably be thankful it does.

Of course, no compiler can offer full protection against programmer sleepiness. In one early C# program I wrote, I defined a *bool* variable named *trigger*, but instead of writing the statement

```
if (trigger)
```

for some reason I wanted to be a little more explicit and wanted to type this:

```
if (trigger == true)
```

Instead, I typed this instead:

```
if (trigger = true)
```

If *trigger* is defined as a *bool* this is a perfectly valid statement in C# but obviously didn't do what I wanted.

If more than one statement should be executed, you can group them as a block of statements in curly brackets:

```
if (a < 5)
{
    b += 27;
    c = 0;
}
```

Some programmers prefer putting the first curly bracket at the end of the line containing the *if* keyword; that's allowed, of course.

The *if* statement can include an *else* clause:

```
if (a < 5)
    b += 27;
else
    b -= 7;
```

Even if the *if* or *else* clause is followed by single statements, some programmers prefer enclosing the single statement in curly brackets.

```
if (a < 5)
{
    b += 27;
}
else
{
    b -= 7;
}
```

You can nest *if* statements, and it's often common that an *else* clause consists of nothing but another entire *if* statement:

```
if (a < 5)
{
    …
}
else
{
    if (a > 5)
    {
        …
    }
    else
    {
        …
    }
}
```

The statements in the second *else* clause are intended to be executed if *a* equals 5. Because the *if* statement inside the first else clause is a single statement, the curly brackets can be removed and the second *if* keyword can be moved to the same line as the first *else*, as shown in this common form:

```
if (a < 5)
{
    …
}
```

```
else if (a > 5)
{
    …
}
else
{
    …
}
```

The curly brackets and the statements within the curly brackets are called a *block*. (See the *C# Language Specification*, §8.2) You can declare new variables within the block, but they are only visible within the block and all nested blocks.

You can't declare a variable with the same name as one already declared in a parent block, for example:

```
int A;
...
{
    int A;    // Not allowed!
    ...
}
```

The error message on the second declaration is: "A local variable named 'A' cannot be declared in this scope because it would give a different meaning to 'A', which is already used in a 'parent or current' scope to denote something else." Again, this restriction helps avoid common bugs.

However, declaring variables with the same name in sibling blocks *is* allowed:

```
{
    int A;
    ...
}
...
{
    int A;    // No problem!
    ...
}
```

The *switch* and *case* construction in C# has a restriction not present in C. In C and C++ you can do this:

```
switch(a)
{
case 3:
    b = 7;
                // Fall through isn't allowed in C#
case 4:
    c = 3;
    break;
```

```
default:
    b = 2;
    c = 4;
    break;
}
```

In C or C++, in the case where *a* is equal to 3, one statement is executed and then execution falls through to the case where *a* is equal to 4. That may be what you intended, or you may have forgotten to type in a *break* statement. To help you avoid bugs like that, the C# compiler will report an error. C# allows a case to fall through to the next case only when the case contains no statements. This is allowed in C#:

```
switch (a)
{
case 3:
case 4:
    b = 7;
    c = 3;
    break;

default:
    b = 2;
    c = 4;
    break;
}
```

To compensate for the restriction against fall-through, C# allows you to use a *goto* statement at the end of a case to branch to another case. This is a legal C# implemention of the illegal *switch* block shown earlier:

```
switch(a)
{
case 3:
    b = 7;
    goto case 4;

case 4:
    c = 3;
    break;

default:
    b = 2;
    c = 4;
    break;
}
```

You don't need the final *break* at the end of a *case* if the *goto* is there instead. You can also branch to the default *case*:

```
switch(a)
{
case 1:
    b = 2;
    goto case 3;
```

```
case 2:
    c = 7;
    goto default;

case 3:
    c = 5;
    break;

default:
    b = 2;
    goto case 1;
}
```

The expression in the *switch* statement must resolve to any integer type, *char*, *string*, or an enumeration, and must match the type in the case labels.

You can indeed use a *string* variable in the *switch* statement and compare it to literal strings in the *case* statements:

```
switch (strCity)
{
case "Boston":
    …
    break;

case "New York":
    …
    break;

case "San Francisco":
    …
    break;

default:
    …
    break;
}
```

Of course, this is exactly the type of thing that causes performance-obsessed C and C++ programmers to cringe. All those *string* comparisons simply *cannot* be very efficient. In fact, because of a technique known as *string interning* (which involves a table of all the unique strings used in a program), it's a lot faster than you might think.

Iteration Statements

C# also supports the *while* statement for repeating a group of statements while a condition is *true*. You can test a conditional at the top of a block

```
while (a < 5)
{
    …
}
```

or at the bottom of a block:

```
do
{
    …
}
while (a < 5);
```

As with the *if* statement, the expression in parentheses must resolve to a *bool*. In the second example, the block is executed at least once regardless of the value of *a*.

The *while* or *do* block can contain a *break* statement, in which case execution continues with the first statement after the *while* or *do* block. The block can also contain a *continue* statement, which skips the remainder of the statements and goes back to the top.

The *for* statement looks the same as in C and C++:

```
for (i = 0; i < 100; i++)
{
    …
}
```

Within the parentheses, the first part is an initializer that's executed before anything in the loop. The second part is a Boolean expression. The contents of the block are executed only if that expression is *true*. The last part is executed at the end of the block. If *A*, *B*, and *C* are expressions, the statement

```
for (A; B; C)
{
    …
}
```

is roughly equivalent to:

```
A;
while (B)
{
    …
    C;
}
```

I say "roughly" because the *for* block might contain a *continue* statement to skip the rest of the block and start with the next iteration. However, the *C* expression will still be executed in that case, whereas it would not in the *while* statement. The *for* block can contain a *break* statement to exit the block.

As in C++, it's very common for C# programmers to define the iteration variable right in the *for* statement:

```
for (float a = 0; a < 10.5f; a += 0.1f)
{
    …
}
```

The variable *a* is only valid within the *for* statement.

A handy addition to the iteration statements that C# inherited from C and C++ is the *foreach* statement, which C# picked up from Visual Basic. I'll show you some examples of *foreach* in the Chapter 10, which discusses arrays. The *foreach* statement also works with other types of collections, and with strings. Suppose you wanted to display all the characters of a string named *str*, each on a separate line. With a *for* loop the code looks like this:

```
for (int i = 0; i < str.Length; i++)
    Console.WriteLine(str[i]);
```

The foreach statement is considerably simpler:

```
foreach (char ch in str)
    Console.WriteLine(ch);
```

The parentheses consist of the definition of a variable named *ch* of type *char*; this variable must match the type of the elements in the array or collection. This is followed by the keyword *in* followed by the variable containing the elements. You can use *break* and *continue* statements within a *foreach* block.

Within the *foreach* block, the iteration variable is read-only. This is rather obvious in the case of strings (because strings are enumerable anyway) but in Chapter 10 you'll see that you *cannot* use the *foreach* statement to initialize the elements of arrays.

The *foreach* statement requires a collection that supports a particular method, as laboriously described in the *C# Language Specification*, §8.8.4. In a practical sense, it can be said that the *foreach* statement works with collections that implement the *IEnumerable* interface.

Jump Statements

The *C# Language Specification*, §8.9 defines the category of jump statements as including *break*, *continue*, and *return*. I'll discuss *return* in more detail in Chapter 11. This category also includes *throw*, which I'll describe in Chapter 12, and finally *goto*.

You've already seen how to use the *goto* in a *switch* statement. You can also use *goto* to branch to a label. A label is defined with an identifier followed by a colon:

```
NowhereElseToGo:
```

You can branch to the statement at that label with the statement

```
goto NowhereElseToGo;
```

Labels have scopes just like variables, and the label must be in the same block or a parent block as the *goto* statement. In other words, you can't jump into the middle of a block. You can jump out of a block, but not into a block.

Chapter 9. The Stack and the Heap

We program in high-level languages for several reasons. Perhaps we prefer that the code we write be compilable for multiple processors or platforms. Perhaps we prefer block structure rather than jumps. And perhaps we prefer solving problems strictly through an abstract quasi-mathematical algorithmic language without taking machine architecture into account.

In real life, however, it is rarely possible to design an efficient programming language or programming interface that lets the programmer remain entirely ignorant of machine or system architecture.

Such is certainly the case with C#. C# doesn't require the programmer to mess around with pointers, but that doesn't mean that pointers can be entirely banished from the programmer's mind. In fact, a very important aspect of C# involves the different way in which instances of classes and instances of structures are stored in memory. This difference is summed up in the following statement:

Classes are reference types; structures are value types.

If programming languages had mantras, that would be the C# mantra.

As you probably know, one common form of memory storage is the *stack*. In very simple computer architectures—CP/M, for example, or a .com executable running under MS-DOS—the program code itself sits at the bottom of a 64K block of memory, and the stack pointer is initially set to the very top of the memory block. The assembly language PUSH instruction decrements the stack pointer and places data at the location addressed by the stack pointer. The POP instruction retrieves an item from the stack and increments the stack pointer. The stack also comes into play when a CALL instruction executes. A pointer to the next instruction is pushed on the stack so that a RET instruction can pop the instruction pointer off the stack and resume execution at the instruction following the CALL.

If items A, B, and C are pushed on the stack in that order, then they are popped from the stack in the order C, B, and A. The stack is thus known as a last-in-first-out (LIFO) memory storage. An LIFO storage mechanism is necessary to implement nested function calls. Each function can also use the stack for storing local variables without interfering with other functions. Because the stack traditionally starts at the high end of memory, it can arbitrarily grow in size. The only problem occurs when the stack gets so big it collides with program code.

In today's environments, the word "stack" is used to refer to any LIFO memory storage, and stacks can pretty much grow as large as they want without getting entangled with program code. Stacks don't have to grow from the top of memory, although they are often still envisioned that way.

Every thread of execution running under Windows has its own stack. The stack is used for storing return addresses during function calls, for passing arguments to function calls, and for storing local variables in each function.

C# is similar. Consider a *Main* method in a C# program that declares two *int* variables, a *long* and a *string*:

```
static void Main()
{
    int A, B;
    long C;
    string D;
    ...
}
```

When a method begins execution, memory on the stack is reserved to store all variables declared in the method, and the memory for these variables is freed when the method reaches its end. In this example, the *A* and *B* variables both require 4 bytes of storage on the stack, and the *C* variable requires 8 bytes. But now we have a problem. Exactly how many bytes are needed to store a *string*?

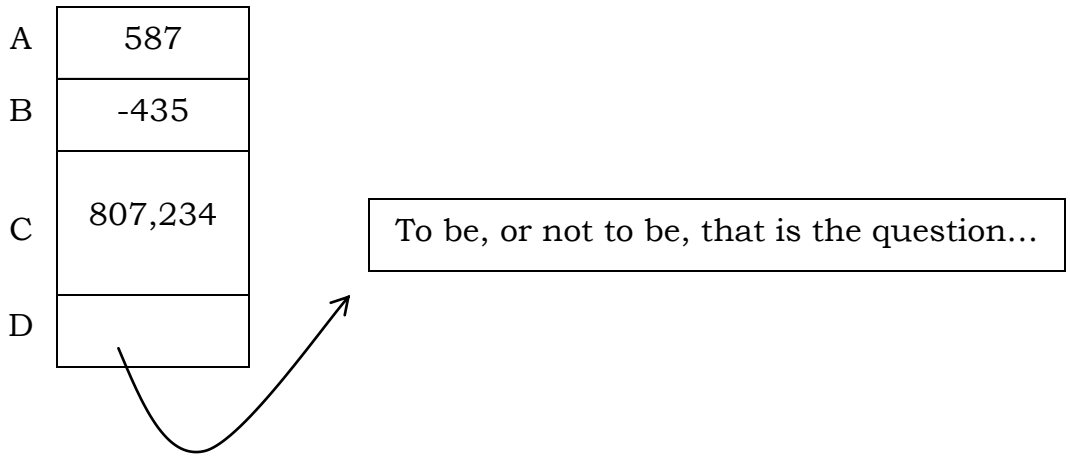
It depends. Strings can be long and strings can be short, and it is simply not possible to anticipate exactly how much memory is required for a particular string, particular when the program can have statements like this:

```
D = Console.ReadLine();
```

For this reason, the string itself is not stored on the stack. Instead, as the program is running, the memory for the string is allocated from an area known as the *heap*, and more specifically, in a *local heap* that is private to the process. The heap is a general-purpose area of storage organized so that chunks of memory of any size can be allocated and freed at random during program execution. (Even nicer is a heap with two levels of indirection that can be compacted if it becomes excessively fragmented.)

Every program running under Windows has its own local heap; the heap is shared among all threads in the program. (The stack and heap are also different in regard to prepositions: We say that something is stored *on* the stack but *in* the heap.)

In this example, the string itself is stored in the heap. However, the *string* variable itself must be stored on the stack. What's actually stored on the stack is a reference to the location of the string in the heap:



The string can be as small or as large as it needs to be, but the space on the stack required for the reference remains the same.

What is this reference exactly? In the C# documentation it's often referred to in a rather vague way, but it's probably something very close to a memory address, and thus not very dissimilar from a traditional pointer. However, there are major differences between references and pointers. The reference is *managed*, and the memory it references is known as the *managed heap*. The program can't manipulate or perform any arithmetic on this reference. And, most importantly, if a memory block allocated from the heap no longer has any references pointing to it, that memory block becomes eligible for garbage collection.

Here's that C# mantra again:

Classes are reference types; structures are value types.

Structures are value types. We've already encountered several structures. All the numeric types in C# are aliases for structures in the .NET Framework. The C# value types include *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, *decimal*, *char*, and *bool*. When you declare variables of these types, the actual numeric values are stored on the stack.

Classes are reference types. The *string* type is an alias for the *System.String* class. It's a class. A reference type is stored on the stack as a reference to an area of memory allocated from the heap.

Classes are reference types; structures are value types.

This difference was established for purposes of efficiency. Heap allocations take time, particularly when there's no more memory in the heap, and the heap manager needs to compact the heap or begin garbage collection. It makes no sense to store something as small as an *int* in the heap when you'll probably need as much memory on the stack to store

the reference as to store the *int* itself. The stack lets an *int* value be stored quickly and retrieved quickly.

But the stack is not appropriate for objects that might vary in size, such as *string* variables. The stack is certainly not appropriate for arrays, which I'll cover in the next chapter. And the stack is not quite suited for some aspects of object-oriented programming. It is often convenient to refer to an instance of a particular class as an instance of an ancestor class. This works best when the object is actually a reference to memory in the heap rather than an area on the stack.

It's possible for a string literal to consist of two quote signs with no characters between them:

```
D = "";
```

A string with no characters is referred to as an *empty* string. Even though the string has no characters, memory is still allocated from the heap to store information about the string, including the fact that it has no characters. The *Length* property would reveal this fact: The expression *D.Length* would equal 0.

If *D* were first assigned a return value from *Console.ReadLine* (as shown in the code earlier in this chapter) and then assigned an empty string, what happens to the memory allocated from the heap for the original string? That string is still in the heap, but there's no longer any reference to it, which means it's taking up unnecessary space that could be used for something else. It becomes eligible for garbage collection. This memory will not be freed immediately, but sometime in the future as the program is running, and particularly if memory gets scarce, the unreferenced memory in the heap will be freed and made available for future allocations.

You can also assign *D* a special value, which is the keyword *null*:

```
D = null;
```

The keyword *null* essentially means "no reference." Instead of storing a reference to something in the heap, the variable *D* is now storing a zero value. A string variable that does not reference any heap memory is called a *null* string. If you tried to determine the length of such a string using *D.Length*, you'd raise a *NullReferenceException*. There is nothing to determine the length of.

You can use *null* with equality and inequality operators. If D is a *null* string, then then the expression

```
D == null
```

is *true*.

The code shown at the beginning of this chapter declared a *string* variable without setting it to anything:

```
string D;
```

In this case, space has been reserved on the stack for the variable *D*, but it is considered to be uninitialized. It doesn't even equal *null* at this point.

The *null* string and the empty string may initially seem pretty similar but you can see now that they're quite different. When a *string* variable equals *null*, the value stored on the stack is 0, which doesn't reference any memory in the heap. When a *string* variable equals the empty string, memory has been allocated from the heap to store the string, but the string is 0 characters in length.

You cannot set a value type (such as an *int* or a *double*) to *null*. Null values only make sense for reference types. (However, C# 2.0 lets you define "nullable" value types, as I'll discuss in Chapter 28.)

Chapter 10. Arrays

Arrays are ordered collections of objects of the same type. Each object in the array is referred to as an *element* of the array. A particular element is associated with an index of the array, which is an integer ranging from zero to one less than the size of the array. (In other words, C# has zero-based array indexing.)

C# lets you define an array of any type. The simplest array declaration is a one-dimensional array. This code defines *A* to be an array of type *int*:

```
int[] A;
```

In an array declaration statement, the square brackets follow the element type, and they must be empty. *A* is a reference and space on the stack is reserved for storing *A*. However, at this point A is uninitialized. Because A is a reference, you can set it to *null*:

```
A = null;
```

Now *A* is no longer uninitialized, but no memory has been allocated for storing the elements of the array. Allocating memory for the array requires a *new* expression, which looks like this:

```
A = new int[100];
```

Another set of square brackets appear, but now they contain the desired number of elements in the array. The *new* expression here allocates sufficient memory in the managed heap for 100 32-bit integers and returns a reference to that memory block, which is then stored in *A*.

You can declare the array variable and initialize it in the same statement:

```
int[] A = new int[100];
```

Memory allocated from the heap is always initialized to zero. Thus, all the elements of the array have an initialized value of zero.

You can reference one of the elements in the array using the array name and an index in square brackets:

```
Console.WriteLine(A[55]);
```

Because all the array elements have been initialized to zero, this statement will display the value zero. If you use a negative index, or an index equal to or greater than the number of elements in the array, you'll raise an *IndexOutOfRangeException*.

You can fill the elements of an array with whatever means is convenient. A *for* loop is common:

```
for (int i = 0; i < 100; i++)
    A[i] = i * i;
```

Any array is implicitly an object of type *System.Array*, and you can use properties and methods defined by the *Array* class with arrays. Perhaps the most important property of the *Array* class is *Length*, which indicates the number of elements in the array. Here's a more generalized way to fill an integer array with squares:

```
for (int i = 0; i < A.Length; i++)
    A[i] = i * i;
```

In this example, *A.Length* is the value 100. As with the same-named property of the *String* class, the *Length* property is read-only.

You can access all the elements of an array sequentially in a *foreach* statement:

```
foreach (int i in A)
    Console.WriteLine(i);
```

However, you cannot initialize the elements of an array in a *foreach* statement because you need to set the value of the iteration variable and that's not allowed.

Later on in the program, you might set *A* to a different *new* expression:

```
A = new int[20];
```

Enough memory is allocated from the managed heap for 20 elements, and the reference to that memory is returned and stored in *A*.

But what happens to the original block of memory that was allocated for the 100 integers? There is no *delete* or *free* operator in C#. If the original block of memory is no longer referenced by anything else in the program, it becomes eligible for garbage collection. At some point, the Common Language Runtime will free up the memory originally allocated for the array.

The number of elements in an array can be determined at runtime:

```
Console.Write("Enter the array size: ");
int num = Int32.Parse(Console.ReadLine());
double D = new double[num];
```

Obviously, this feature eliminates any need for C-type memory allocation functions such as *malloc*.

In some cases, when writing a program you might know both the size of the array and the elements it should contain. In that case you can initialize the array when creating it:

```
double[] D = new double[3] { 3.14, 2.17, 100 };
```

The number of initializers must equal the declared size of the array. If you're initializing the array, you can leave out the size of the array because it's determined from the number of initializers:

```
double[] D = new double[] { 3.14, 2.17, 100 };
```

Notice the empty square brackets in the *new* expression. You can even leave out the *new* expression:

```
double[] D = { 3.14, 2.17, 100 };
```

This shortcut is available only in the statement that declares the array variable. You can't do something like this:

```
double[] D;
D = { 3.14, 2.17, 100 };    // Won't work!
```

It's possible to declare multiple array variables in a single declaration statement:

```
decimal[] sales, commissions, bonuses;
```

You can allocate memory for all of these arrays, or only some of them, and even initialize elements of one or more of the arrays in the same declaration statement, but it might get so messy that you'll want to split it into several declarations.

Here's a declaration statement that allocates memory for an array of strings:

```
string[] strs = new string[10];
```

That *new* expression allocates enough memory from the heap for ten strings. However, because *String* is a class, and classes are reference types, the *new* expression really allocates enough memory from the heap for ten *references*. Heap memory is always initialized to zero, which means that the ten elements of the array are effectively set to *null*. The expression

```
strs[5] == null
```

returns *true*. Although these references are initially *null*, eventually they will probably reference actual strings that are themselves stored in the heap.

Understanding array creation and initialization is sometimes clarified if you look at it in terms of expressions. The expression

```
new string[4]
```

allocates enough memory for an array of four strings and returns a reference to that memory. If the element type were a value type (such as *int* or *decimal*), each element would be initialized to 0. Because the element type here is a reference type (specifically *string*), each element is initialized to *null*.

Here's another expression:

```
new string[]{ "North", "East", "South", "West" }
```

This expression allocates memory for an array of four strings. Each element of the array is another reference to the actual string.

Now here's an interesting expression:

```
new string[] { "North", "East", "South", "West" } [2]
```

It's the same as the previous expression except that it also includes an array index at the end. If you check the Operator Precedence and Associativity table at the back of this book, you'll see that the keyword *new* and array indexing (indicated by *a[x]*) have the same precedence and are associated from left to right. The index essentially chooses one of the elements of the array. This expression evaluates to the string "South".

This means that if you need to use a little array just once, you don't need to declare an array variable. Here's a statement that displays one of the four compass points based on the variable *dir*:

```
Console.Write(new string[] { "North", "East", "South", "West" } [dir]);
```

The array is created in the course of the execution of this statement and then becomes eligible for garbage collection. I certainly would avoid putting a statement like this in any kind of loop or a method that's frequently called, but for a one-time execution, it's certainly elegant.

C# provides two types of multi-dimensional arrays. The simplest type of multi-dimensional array is declared using a single set of square brackets, and commas indicate multiple dimensions. Here are declarations of one-dimensional, two-dimensional, and three-dimensional arrays:

```
int[] one;
int[,] two;
int[,,] three;
```

You can allocate memory for these arrays using *new* expressions that contain the size of each dimension separated by commas:

```
one = new int[15];
two = new int[3, 6];
three = new int[8, 5, 3];
```

Or, you could include the *new* expression in the declaration.

You reference an element of the array with indices separated by commas:

```
three[i, j, k] = 39;
```

In this example the variable *i* must be in the range 0 through 7, *j* must be 0 through 4, and *k* must be 0, 1 or 2.

The *Length* property reports the total number of elements in the array, which is equal to the product of the sizes of each dimension. For example, *three.Length* returns 120. The *Rank* property indicates the number of dimensions: *three.Rank* returns 3.

If you need the number of elements in a particular dimension, you can use the *GetLength* method of the *Array* class. The argument is a zero-based dimension. For example,

```
three.GetLength(1)
```

returns the size of the second dimension of *three*, which is 5. The *Array* class also contains methods for sorting and searching arrays.

Multidimensional arrays seem to be less common in object-oriented programming than in traditional procedural programming. It's probably more common in object-oriented programming to have single-dimensional arrays of objects, where the objects themselves encapsulate multiple items. But some "real-life" examples of multidimensional arrays do exist. If you were unfortunate enough to be working on a program involving United States senators, for example, the following array would help store their names:

```
string[,] senators = new string[50,2];
```

That's 50 states and 2 senators each.

Initializing the elements of multidimensional arrays in a *new* expression requires a precise use of curly brackets. Here's another three-dimensional array that's a bit smaller than the previous one:

```
int[,,] arr = new int[3, 2, 4] {{{ 8, 3, 4, 2}, { 7, 4, 1, 2}},
                                {{ 2, 7, 3, 6}, { 5, 1, 9, 0}},
                                {{ 0, 4, 9, 7}, { 3, 9, 8, 5}}};
```

The first four initialization values are *arr[0,0,0]* through *arr[0,0,3]*, the second four values are *arr[0,1,0]* through *arr[0,1,3]*, and so forth. The last four values are arr[2,1,0] through arr[2,1,3]. The *Rank* property of this array returns 3; the *Length* property returns 24. You can shorten the array initialization to

```
int[,,] arr = new int[,,] {{{ 8, 3, 4, 2}, { 7, 4, 1, 2}},
                           {{ 2, 7, 3, 6}, { 5, 1, 9, 0}},
                           {{ 0, 4, 9, 7}, { 3, 9, 8, 5}}};
```

without explicitly specifying the number of elements in each dimension, or you can leave out the *new* expression entirely:

```
int[,,] arr = {{{ 8, 3, 4, 2}, { 7, 4, 1, 2}},
               {{ 2, 7, 3, 6}, { 5, 1, 9, 0}},
               {{ 0, 4, 9, 7}, { 3, 9, 8, 5}}};
```

C# also supports arrays of arrays, which are essentially arrays in which the elements are themselves arrays. These are referred to as *jagged arrays* because the size of the second dimension (and possibly subsequent dimensions) is not constant. The size of each dimension is different depending on the index of the previous dimension.

For example, suppose you want to declare an array for storing all the family members of your four closest friends. These families range in size

from two people to eight people. You could certainly declare a normal two-dimensional array sufficient for the largest family:

```
string[,] normalArray = new string[4, 8];
```

But that approach wastes some space. Not all of your four friends' families have eight people. (And if the wasted space of this example seems meager, how about an array similar to the *senators* array but for members of the House of Representatives? Depending on the state, the number of representatives ranges from 1 to 53.)

Because a jagged array is essentially an array of arrays, creating the array requires multiple *new* expressions. Here's the declaration and the first *new* expression for the array that stores your four friends' family members:

```
string[][] jaggedArray = new string[4][];
```

Notice the use of multiple square brackets. Next you need four additional new statements for each of the four families:

```
jaggedArray[0] = new string[5];
jaggedArray[1] = new string[2];
jaggedArray[2] = new string[8];
jaggedArray[3] = new string[4];
```

Each of these *new* expression indicates the number of members in that family. The family sizes range from two to eight.

At this point, you can access *jaggedArray[0][0]* through *jaggedArray[0][4]* for the five members of the first family. The two members of the second family are stored in *jaggedArray[1][0]* and *jaggedArray[1][1]*. And so forth.

An assignment such as

```
jaggedArray[3] = new string[4];
```

shown above can also include initializations for that family:

```
jaggedArray[3] = new string[4] { "Jack", "Diane", "Bobby", "Sally" };
```

or, a tiny bit simpler,

```
jaggedArray[3] = new string[] { "Jack", "Diane", "Bobby", "Sally" };
```

You can also initialize the whole array in the original declaration. The initialization includes all the *new* expressions:

```
string[][] jaggedArray = new string[4][]
    {
        new string[] { "Jill", "Alice", "Billy", "Judy", "Sammy" },
        new string[] { "James", "Ellen" },
        new string[] { "Steve", "Sue", "Bernie", "Rich",
                       "Chris", "Erika", "Michelle", "Alyssa" },
        new string[] { "Jack", "Diane", "Bobby", "Sally" }
    };
```

Because this is a declaration, the first *new* expression can be eliminated, but the rest are required.

Chapter 11. Methods and Fields

When a certain block of code needs to be executed multiple times while a program is running, it is common to put it in a loop. If the same block of code must be executed from different parts of the program, it is common to isolate it in a unit called in various languages a subroutine or function, but which in C# is called *method*. Every C# program must contain a method named *Main*. All but the most trivial C# programs contain additional methods as well.

For example, suppose you need to write a program that asks the user to type in some information, perhaps a first name, last name, and age. Each of the three items requires a call to *Console.Write* to displays a prompt such as "Enter your first name." Each of the three items requires a call to *Console.ReadLine* to obtain the information the user types. Here's an approach where everything is in the *Main* method

Interrogation1.cs

```
//---------------------------------------------
// Interrogation1.cs (c) 2006 by Charles Petzold
//---------------------------------------------
using System;

class Interrogation1
{
    static void Main()
    {
        Console.Write("Enter your first name: ");
        string strFirstName = Console.ReadLine();

        Console.Write("Enter your last name: ");
        string strLastName = Console.ReadLine();

        Console.Write("Enter your age: ");
        string strAge = Console.ReadLine();

        Console.WriteLine();
        Console.WriteLine("First name: " + strFirstName);
        Console.WriteLine("Last name: " + strLastName);
        Console.WriteLine("Age: " + strAge);
    }
}
```

Of course, in a real program, the age wouldn't be stored in a *string*. It would be stored as an integer, which means there'd be a call to *Int32.Parse*, but I'm trying to keep this simple.

Rather than calling *Console.Write* three times to display a prompt, and *Console.ReadLine* three times to obtain what the user types, it would be more convenient for the *Main* method to call another method three times. Perhaps this method is called *GetInfo*. The first part of *Main* might be simplified to look something like this:

```
string strFirstName = GetInfo("Enter your first name: ");
string strLastName = GetInfo("Enter your last name: ");
string strAge = GetInfo("Enter your age: ");
```

Here's a *GetInfo* method that does the grunt work:

```
static string GetInfo(string strPrompt)
{
    Console.Write(strPrompt);
    string strReturn = Console.ReadLine();
    return strReturn;
}
```

Like *Main*, the *GetInfo* method must be static. (Non-static methods will make their appearance in Chapter 13.) Following the *static* keyword is *string*, which is the return value of the *GetInfo* method. If a method has a return type other than *void*, all code paths within the method must terminate with a *return* statement that returns an object of the proper type. (A method with a *void* return type can have a *return* statement by itself to end execution of the method.) The parameter is also a string, called *strPrompt* within the method.

The *GetInfo* method displays the prompt, and stores the return value of *Console.ReadLine* in *strReturn*, which it then returns. As with C and C++, the return value does not have to be in parentheses, although many programmers tend to use them:

```
return (strReturn);
```

The *strReturn* variable is local to *GetInfo*, and is only visible within *GetInfo* after its declaration. The method could actually be simplified a little by combining the last two statements and eliminate the *strReturn* variable entirely:

```
return Console.ReadLine();
```

Perhaps you're not quite sure you want to terminate each of the prompts with a colon. Perhaps you suspect you may want to change it to a little arrow. In that case, you might write the first statement of *GetInfo* like this:

```
Console.Write(strPrompt + ": ");
```

And call *GetInfo* like this:

```
string strFirstName = GetInfo("Enter your first name");
```

Here's the whole program with these changes.

Interrogation2.cs

```
//-----------------------------------------------
// Interrogation2.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class Interrogation2
{
    static void Main()
    {
        string strFirstName = GetInfo("Enter your first name");
        string strLastName = GetInfo("Enter your last name");
        string strAge = GetInfo("Enter your age");

        ShowInfo(strFirstName, strLastName, strAge);
    }
    static string GetInfo(string strPrompt)
    {
        Console.Write(strPrompt + ": ");
        return Console.ReadLine();
    }
    static void ShowInfo(string strFirstName, string strSurName,
                         string strYearsOld)
    {
        Console.WriteLine("First name: " + strFirstName);
        Console.WriteLine("Last name: " + strSurName);
        Console.WriteLine("Age: " + strYearsOld);
    }
}
```

I've also added a second method called *ShowInfo* that displays the information. Although *ShowInfo* is only called once from *Main*, that doesn't necessarily prohibit it from being a separate method. *ShowInfo* has three parameters but a *void* return value.

The order of methods in a class doesn't matter. Methods don't have to be declared before they are referenced.

Two of the parameters to *ShowInfo* have slightly different names than the string variables defined in *Main*. It doesn't matter whether they're the same or different. The variables defined in *Main* aren't visible in *ShowInfo*, and the *ShowInfo* parameters aren't visible in *Main*.

Both *GetInfo* and *ShowInfo* are static. The *Main* method can refer to these methods by just their names, but it could also preface the method name with the class name:

```
string strFirstName = Interrogation2.GetInfo("Enter your first name");
```

ShowInfo has a parameter for each of the items it needs to display. There's nothing really wrong with this approach until you want to start adding more items to the list of information you obtain. For each new item you'll need another string variable, of course, and another call to *GetInfo*, and you'll need to add a line to *ShowInfo* to display the new item.

You'll also need to add another parameter to *ShowInfo*, which means you'll have to change the method itself, and the call to that method.

And what if it gets to the point where you're asking the user for twenty pieces of information? You'll need to have twenty arguments to *ShowInfo* and you have to be very careful that your call to *ShowInfo* has all the arguments in the correct order.

Another approach would be to save the user's responses as fields. As you know, fields look like local variables, except they are not defined inside a method. They are defined inside the class but outside of all methods, and they can be accessed by any method in the class.

Interrogation3.cs

```
//-----------------------------------------------
// Interrogation3.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class Interrogation3
{
    static string strFirstName, strLastName, strAge;

    static void Main()
    {
        strFirstName = GetInfo("Enter your first name");
        strLastName = GetInfo("Enter your last name");
        strAge = GetInfo("Enter your age");

        ShowInfo();
    }
    static string GetInfo(string strPrompt)
    {
        Console.Write(strPrompt + ": ");
        return Console.ReadLine();
    }
    static void ShowInfo()
    {
        Console.WriteLine();
        Console.WriteLine("First name: " + strFirstName);
        Console.WriteLine("Last name: " + strLastName);
        Console.WriteLine("Age: " + strAge);
    }
}
```

Notice the three fields of type *string* declared at the top of the *Interrogation3* class. Although it's common to put fields at the top of a class, they don't need to be there; they can just as well be at the bottom of the class after all the methods, or even between two methods, or scattered among the methods.

Now *ShowInfo* has no parameters, and the three statements in *ShowInfo* refer to the fields set in the *Main* method. With each additional item,

you'll need to make three changes: a new field, a new call to *GetInfo*, and a new line in *ShowInfo*.

You're still not sure about the colon at the end of the prompt, but you know that if you change it, you'll also want to change the three statements in *ShowInfo* likewise. It would be nice to define this character sequence just once, and use it wherever needed. You could make that a field as well:

```
static string strDelimiter = ": ";
```

And then you can change the *Console.Write* statement in *GetInfo* and the three statements in *ShowInfo* like so:

```
Console.WriteLine("First name" + strDelimiter + strFirstName);
```

This is pretty good, but there's another change you might consider. The field named *strDelimiter* is never changed during the time the program is running. In fact, you want to make sure that it's never changed. You want to prevent yourself (or someone else) from modifying the program and adding code that inadvertently changes *strDelimiter*. One way to do this is to add a modifier named *readonly*:

```
static readonly string strDelimiter = ": ";
```

The order of the *static* and *readonly* keywords doesn't matter, but they both must precede the type of the field, which is *string*. With the *readonly* modifier, any code that tries to change the value of *strDelimiter* will be flagged by the C# compiler as an error. (That's not entirely true. A constructor can change the value, as you'll see in Chapter 17.)

Another possibility is this:

```
const string strDelimiter = ": ";
```

A constant must be initialized in its declaration statement. A *const* is implicitly static.

There is a big difference between *const* and *static readonly*: A *const* is evaluated during compilation and the value is substituted wherever it's used. A *static readonly* field is evaluated at runtime. But in practice they're pretty much the same.

In C and C++ you can have a local variable defined as *static*, and that variable will retain its value between function calls. That option is not available in C#. You can have a local *const*, however, which is a constant whose visibility is restricted to a method. But the *readonly* modifier is applicable only for fields, and can't be used for local variables.

In C#, as in C++, you can have multiple methods with the same name. These are known as overloads. These multiple methods must be distinguished by having a different number of arguments, or arguments with different types. In C#, you can't have two methods in the same class with the same name that differ only by the return type.

Here's yet another version of the Interrogation program that has a parameterless method named *GetInfo* that calls the parametered versions of *GetInfo*:

Interrogation4.cs

```
//-----------------------------------------------
// Interrogation4.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class Interrogation4
{
    const string strDelimiter = ": ";
    static string strFirstName, strLastName, strAge;

    static void Main()
    {
        GetInfo();
        ShowInfo();
    }
    static void GetInfo()
    {
        strFirstName = GetInfo("Enter your first name");
        strLastName = GetInfo("Enter your last name");
        strAge = GetInfo("Enter your age");
    }
    static string GetInfo(string strPrompt)
    {
        Console.Write(strPrompt + strDelimiter);
        return Console.ReadLine();
    }
    static void ShowInfo()
    {
        Console.WriteLine();
        Console.WriteLine("First name" + strDelimiter + strFirstName);
        Console.WriteLine("Last name" + strDelimiter + strLastName);
        Console.WriteLine("Age" + strDelimiter + strAge);
    }
}
```

Sometimes the choice of which overloaded method to call can be tricky for the C# compiler. For example, suppose there are two methods with the same name, but one has a *long* parameter and the other has a *double* parameter. Some code calls the method with an *int* argument. Which method does C# choose? Overload resolution is described in the *C# Language Specification*, §7.4.2.

The next step in this Interrogation series of programs might be to define a new class named *Person* with instance fields of *strFirstName*, *strLast-Name*, and *strAge*, and jump right into object-oriented programming. But let's hold off on that for another chapter or so, and explore some other method-related issues.

Arguments to methods are normally passed by value, which means that the following program displays 22 rather than 55:

PassByValue.cs

```
//-----------------------------------------
// PassByValue.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class PassByValue
{
    static void Main()
    {
        int i = 22;
        AlterInteger(i);
        Console.WriteLine(i);
    }
    static void AlterInteger(int i)
    {
        i = 55;
    }
}
```

When a method such as *AlterInteger* is executing, its stack contains space for all the parameters to the method (in this case just one *int*) and all its local variables (in this case, none). As *AlterInteger* is called from *Main*, a copy of the argument passed to *AlterInteger* is stored on the stack for *AlterInteger* to use. *AlterInteger* refers to a copy of the integer rather than to the integer referenced by *Main*.

However, there are times when you would rather be able to pass a number or other object to a method by reference, so that any changes made to the object within the method are reflected in the value after the method has ended.

You can do this using the *ref* keyword, as demonstrated in the following program. This program displays 55.

PassByReference.cs

```
//-------------------------------------------------
// PassByReference.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;

class PassByReference
{
    static void Main()
    {
        int i = 22;
        AlterInteger(ref i);
        Console.WriteLine(i);
    }
```

```
    static void AlterInteger(ref int i)
    {
        i = 55;
    }
}
```

Notice that the *ref* keyword is required both in the declaration of the method and when the method is called. This double requirement is just to prevent you, the programmer, from making mistakes.

Inside *AlterInteger*, the stack contains a reference rather than the integer itself. That reference is to the original integer in *Main*.

The C# language also supports a similar keyword named *out*. (You saw out in the Math.DivRem method in Chapter 6. You'll see it again in the TryParse method in Chapter 12.) When compiled to intermediate language, the *ref* and *out* keywords are identical. However, when you use *ref*, the C# compiler requires that the variable whose reference you're using has already been initialized. When using *out*, the variable doesn't have to be initialized.

For example, in the preceding program, if you change

```
int i = 22;
```

to

```
int i;
```

then the program will no longer compile because the variable has not been initialized when the method is called. If you change the two occurrences of *ref* to *out*, C# will obligingly compile the code. However, now try changing *AlterInteger* to this:

```
static void AlterInteger(out int i)
{
    i += 33;
}
```

This has now become unacceptable to the C# compiler because the body of the method implies that the parameter has already been set before the method is called.

If you don't use the *ref* or *out* keywords, then arguments to methods are passed by value. Essentially, a copy of the object is made and placed on the stack for the method to use. If this object is a reference type (that is, an array, or an instance of a class) then a copy of the reference is made for use by the method.

So, if a parameter is a reference type, then the value passed to the method is actually a reference, as the following program demonstrates.

PassArrayByValue.cs

```
//-----------------------------------------------
// PassArrayByValue.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class PassArrayByValue
{
    static void Main()
    {
        int[] arr = new int[10];
        arr[0] = 22;
        AlterInteger(arr);
        Console.WriteLine(arr[0]);
    }
    static void AlterInteger(int[] arr)
    {
        arr[0] = 55;
    }
}
```

This program creates an array of integers and sets the first element to 22. Then it passes this array to *AlterInteger*. However, that doesn't mean the entire array is copied on the stack. The array is a reference, so only a copy of that reference is made for use by the *AlterInteger* method. Even though it's a copy of the reference, both references access the same heap memory. *AlterInteger* is able to change an element of the original array. The program displays 55.

Now, add the following statement to the bottom of *AlterInteger*:

```
arr = null;
```

The program still works as before. The method is only setting its copy of the *arr* reference to *null*, so *Main* still has its original *arr* reference and nothing bad happens.

Now let's add the *ref* keyword to the method parameter, and see what happens.

PassArrayByReference.cs

```
//--------------------------------------------------
// PassArrayByReference.cs (c) 2006 by Charles Petzold
//--------------------------------------------------
using System;

class PassArrayByReference
{
    static void Main()
    {
        int[] arr = new int[10];
        arr[0] = 22;
        AlterInteger(ref arr);
```

```
        Console.WriteLine(arr[0]);
    }
    static void AlterInteger(ref int[] arr)
    {
        arr[0] = 55;
        arr = new int[5];
    }
}
```

Now the *arr* reference itself declared in *Main* is passed by reference, so that the method is able to alter the *arr* itself in *Main* rather than just elements of the array. The method sets *arr* to a new array, and each element of a nearly allocated array has an initialized value of 0. The program displays the number 0.

The difference between value types and reference types in C# may take some getting accustomed to. Experimentation is encouraged. I'll explore this topic more in Chapter 16.

The previous two program demonstate that you can pass whole arrays to methods. The methods can then determine the size of the arrays with the *Length* property and examine or change the array elements. However, a method declared with an array parameter can be a bit more versatile with the *params* keyword, as the following program demonstrates.

PassParamsArray.cs

```
//-----------------------------------------------
// PassParamsArray.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class PassParamsArray
{
    static void Main()
    {
        int[] arr = { 22, 33, 55, 100, 10, 2 };
        Console.WriteLine(AddUpArray(arr));
        Console.WriteLine(AddUpArray(22, 33, 55, 100, 10, 2));
    }
    static int AddUpArray(params int[] arr)
    {
        int sum = 0;

        foreach (int i in arr)
            sum += i;

        return sum;
    }
}
```

Without the *params* keyword, the method would be able to accept an argument that is an array of integers, add up the elements, and return the sum. With the keyword *params*, another option becomes available: A

list of integers can simply be passed to the method, as demonstrated by the second call to *AddUpArray* (in the second *WriteLine*).

Behind the scenes, the list of integers is made into an array, so there's definitely no performance advantage to providing a variable list of arguments. There must be no more than one *params* parameter to a method, and it must be the last parameter. These rules are obvious if you consider the confusion that might result without them.

The most generalized form of *Console.WriteLine* and *String.Format* use a *params* parameter for the list of objects after the formatting string. Both classes also provide methods with one, two, or three objects following the formatting string to prevent behind-the-scenes array creation when only a couple items are being formatted.

As I've discussed, when a method begins execution, space on the stack is reserved for all the local variables in the method. For an array, the amount of space needed on the stack is the size of a reference. If the declaration of the array also includes element initialization, the compiler generates code that allocates memory from the heap and initializes all the element values in the array one by one.

If the array is declared in *Main*, the array creation and initialization occur once when the program starts up. But suppose the array is in another method. Here's a little method that calculates a day-of-year value based on one-based month and day parameters (ignoring leap years);

```
static int DayOfYear(int month, int day)
{
    int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                             181, 212, 243, 273, 304, 334 };

    return cumulativeDays[month - 1] + day;
}
```

For example the expression

```
DayOfYear(5, 10)
```

returns the day-of-year value for May 10th, which is 130.

A program might be going through a file and calling this method hundreds or thousands or even millions of times. Each and every time the *DayOfYear* method is called, the array must be allocated from the heap and initialized. After the method exits, the block of memory allocated from the heap is no longer referenced and becomes eligible for garbage collection.

The whole process seems to cry out for some kind of constant array. However, you can't declare an array using *const*. Constants can be set only to values available at compile time. An array requires a *new* operation, and *new* operations occur at runtime.

What you *can* do, however, is define the array as *static*, which means that it is initialized only once. But then you can't have the array inside the method in which it's used. The array must be a field.

The following program defines two *DayOfYear* methods, one of which uses an array inside the method, and another which uses a static array outside the method. The program also uses the *Random* class to generate random numbers, and the *Stopwatch* class to measure the time it takes for 10 million calculations to occur.

The *Stopwatch* class in this project is in the *System.Diagnostics* namespace, and it's in the System assembly, which is System.dll, so you'll need a reference to that library. In the Solution Explorer in Visual Studio, right click References, select Add Reference from the menu, and find System.

TestArrayInitialization.cs

```
//-------------------------------------------------------
// TestArrayInitialization.cs (c) 2006 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Diagnostics;

class TestArrayInitialization
{
    const int iterations = 10000000;

    static int[] daysCumulativeDays = { 0, 31, 59, 90, 120, 151,
                                        181, 212, 243, 273, 304, 334 };

    public static void Main()
    {
        Stopwatch watch = new Stopwatch();
        Random rand = new Random();

        watch.Start();

        for (int i = 0; i < iterations; i++)
            DayOfYear1(rand.Next(1, 13), rand.Next(1, 29));

        watch.Stop();
        Console.WriteLine("Local array:  " + watch.Elapsed);

        watch.Reset();
        watch.Start();

        for (int i = 0; i < iterations; i++)
            DayOfYear2(rand.Next(1, 13), rand.Next(1, 29));

        watch.Stop();
        Console.WriteLine("Static array: " + watch.Elapsed);
    }
```

```
    static int DayOfYear1(int month, int day)
    {
        int[] daysCumulative1 = { 0, 31, 59, 90, 120, 151,
                                  181, 212, 243, 273, 304, 334 };

        return daysCumulative1[month - 1] + day;
    }

    static int DayOfYear2(int month, int day)
    {
        return daysCumulativeDays[month - 1] + day;
    }
}
```

The array defined inside the method is faster than you might anticipate. Fortunately array initialization is optimized, so you can keep locally used arrays inside of methods and not worry *too* much.

Chapter 12. Exception Handling

C# supports structured exception handling. As you've experienced, exceptions occur during runtime and are identified by classes, such as the *DivideByZeroException* class. Most of the basic exception classes can be found in the *System* namespace; some are explicitly listed in the *C# Language Specification*, §16.4. Often the documentation of various classes, methods, and properties indicate exactly what exceptions can be raised.

For example, the static *Double.Parse* method indicates it can raise three types of exceptions:

- *FormatException* if the argument is not in the correct format.
- *OverflowException* if the resultant number is smaller than *Double.MinValue* or larger than *Double.MaxValue*.
- *ArgumentNullException* if the *string* argument to the method is *null*.

For example, the string "5.45E400" passed to *Double.Parse* would rasie an *OverflowException* because the exponent is too large.

If you'd prefer that such problems are handled gracefully by your program rather than by the Common Language Runtime, you can enclose the call to *Double.Parse* in a *try* block, which is followed by a *catch* clause that deals with the exception:

```
double input;

try
{
    input = Double.Parse(Console.ReadLine());
}
catch
{
    Console.WriteLine("You typed an invalid number");
    input = Double.NaN;
}
```

Notice that *input* variable is declared before the *try* block. If it's declared within the *try* block, then it wouldn't be available outside the *try* block, and couldn't be referred to elsewhere.

If *Double.Parse* succeeds in converting the input string into a *double*, execution continues at the next statement following the *catch* block. If *Parse* throws an exception, the the *catch* block catches it. The code in the catch block is executed, and then normal execution resumes with the code following the *catch* block. This particular *catch* block displays a message and then sets *input* to NaN. (You could also initialize *input* to

NaN in the declaration statement and leave out this assignment.) Presumably, the code that follows the *catch* block checks the value of *input* for a NaN value and requests the user to re-enter the number.

In a real-life program that reads numeric values from the user, you'll probably put the *try* and *catch* blocks in a *do* loop and keep asking the user to re-enter the values until *Parse* properly returns, as the following program demonstrates.

InputDoubles.cs

```
//-----------------------------------------
// InputDoubles.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class InputDoubles
{
    static void Main()
    {
        double num = GetDouble("Enter the base: ");
        double exp = GetDouble("Enter the exponent: ");
        Console.WriteLine("{0} to the power of {1} is {2}",
                    num, exp, Math.Pow(num, exp));
    }
    static double GetDouble(string strPrompt)
    {
        double input = Double.NaN;

        do
        {
            Console.Write(strPrompt);

            try
            {
                input = Double.Parse(Console.ReadLine());
            }
            catch
            {
                Console.WriteLine();
                Console.WriteLine("You typed an invalid number!");
                Console.WriteLine("Please try again.");
                Console.WriteLine();
            }
        }
        while (Double.IsNaN(input));

        return input;
    }
}
```

It's important to recognize that if *Parse* throws an exception, the *Parse* method doesn't actually return to the program the way it normally does. Execution leaps from somewhere deep inside the *Parse* method to the *catch* clause. In this program, if you don't initialize the value of *input* in

the declaration statement (or explicitly set *input* to a value before calling *Double.Parse*), then *input* will be uninitialized in the *catch* block.

The *catch* clause in the InputDouble program is known as a *general catch* clause. It will catch any exception raised in the *try* block. You can instead indicate that a *catch* clause apply only to a specific type of exception. For example, here's a little variation of the basic *try* statement that includes a specific *catch* clause:

```
try
{
    // Statement or statements to try
}
catch (System.Exception exc)
{
    // Error processing
}
```

The *catch* keyword is followed by parentheses and a variable declaration that makes it look a bit like a parameter list to a method. *Exception* is a class in the *System* namespace—you can leave out the *System* preface if you have a *using* directive for *System*, of course—and *exc* (which you can name whatever you want) is declared to be an object of type *Exception*. Within the *catch* block, you can use this *Exception* object to obtain more information about the error. You can display the *Message* property of the object like so:

```
Console.WriteLine(exc.Message);
```

For the InputDouble program, the *Message* property will be either the string

```
Input string was not in a correct format.
```

if the user types letters rather than numbers (for example), or:

```
Value was either too large or too small for a Double.
```

This error can be raised if the number is typed in scientific notation with too large or too small an exponent. You may prefer displaying messages like these to the user rather than making up your own.

If you pass the *Exception* object directly to *WriteLine* as

```
Console.WriteLine(exc);
```

you'll effectively call the *ToString* method of the *Exception* class, which displays detailed information, including a stack trace. This is very useful during program development.

Although the *catch* clause with the *Exception* object is classified as a specific *catch* clause, it's really just as generalized as the general *catch* clause. That's because all the different exception classes (such as *DivideByZeroException* and *OverflowException*) are defined in a class hierarchy with *Exception* at the top. (Actually, as Jeff Richter notes in

CLR via C#, pages 426-427, it's possible for CIL code to throw exceptions not derived from *Exception*.)

Earlier I indicated the various exceptions that *Double.Parse* can raise. You can get very specific in the way you handle each of these exception:

```
try
{
    input = Double.Parse(Console.ReadLine());
}
catch (FormatException exc)
{
    // Handle format exceptions
}
catch (OverflowException exc)
{
    // Handle overflow exceptions
}
catch (ArgumentNullException exc)
{
    // Handle null argument exceptions
}
catch (Exception exc)
{
    // Handle all other exceptions (if any)
}
```

The *catch* clauses are examined in sequence for the first one that matches the exception. The final *catch* clause can alternatively be a general clause with no parameter. At any rate, if you're examining individual types of exceptions, you should always include a general *catch* clause or a *catch* clause using *Exception* at the end to process all the exceptions that are not handled individually.

In this particular example, the penultimate *catch* clauses should never be executed because *Console.ReadLine* never returns *null*. But including non-functional *catch* clauses never hurts, even if it just contains the statement:

```
Console.WriteLine("This statement should never be executed");
```

It's surprising how often you see such messages!

The *try* block in this example actually has two method calls: *Console.ReadLine* is executed first, and then *Double.Parse*. It is possible for *Console.ReadLine* to raise three types of exceptions:

- *IOException* if an input error occurs.
- *OutOfMemoryException* if not enough memory is available to store the *string* object.
- *ArgumentOutOfRangeException* if the number of characters typed by the user is greater than *Int32.MaxValue* or 2,147,483,647.

Of course, it is *extremely* improbable that *Console.ReadLine* will raise any of these three exceptions, and only someone who's extremely obsessive-compulsive (or fatally pessimistic) will enclose every *Console.ReadLine* statement in a *try* block. But it's useful to keep in mind that any allocation of memory might raise an exception.

There's a third clause, called the *finally* clause, that you can use in the *try* statement. The *finally* clause comes after all the *catch* clauses, like this:

```
finally
{
    // Statements in finally block
}
```

The statements in the *finally* clause are guaranteed to execute following the execution of the *try* clause (if no exception is thrown) or the relevant *catch* clause.

At first, the *finally* clause doesn't seem necessary. If you want code executed after *try* and *catch*, why can't you simply put it after the *catch* clause itself?

The answer is simple: The *try* or *catch* clause could contain a *return* statement to return control to the calling method or (if the method is *Main*) to terminate the program. In that case, the statements following the last *catch* clause of the *try* statement would not be executed. That's where the *finally* clause helps out. If the *try* or *catch* clause contains a *return* statement, the statements in the *finally* clause are guaranteed to execute regardless. It's also possible to exit a *try* or *catch* clause with a *goto* or a *throw* statement, which I'll describe shortly. The statements in the *finally* clause execute in those cases as well.

You generally use a *finally* clause for clean up. A *finally* clause might close a file, for example.

It's possible to have a *finally* clause following a *try* clause but with no *catch* clause. In this case, the user is notified of the error as if the program did not handle the exception, but the *finally* clause is executed before the program is terminated.

Besides catching exceptions, you should also know how to throw them. If you're writing a method that might encounter problems of various sorts, generally you'll want the method to throw an exception to notify the code calling the method of these problems. The *throw* statement can be as simple as:

```
throw;
```

But this simple form of the *throw* statement can be used only in a *catch* block to rethrow the exception.

Otherwise, you must supply an argument, which is an instance of the *Exception* class or any class that is derived from *Exception*, including classes you write yourself. Here's the simplest case:

```
throw new Exception();
```

But that's being unnecessarily vague about the type of error that occurred. Try to use one of the more specific exceptions. Very often you'll find a descendent of *Exception* in the *System* namespace that comes close to what you want. For example, if your method has a *string* parameter and the method can't work if a *null* argument is passed, you'll probably have code that looks like this:

```
if (strInput == null)
    throw new ArgumentNullException();
```

It's also possible to pass a *string* argument to the *ArgumentNullException* constructor, perhaps to indicate the particular method parameter that caused the problem:

```
throw new ArgumentNullException("Input string");
```

That string you pass becomes part of the exception message in a *catch* clause. Instead of the *Message* property being

```
Value cannot be null.
```

it will be:

```
Value cannot be null.
Parameter name: Input string
```

As soon as a *throw* statement executes, the method is finished. No further code will be executed. If *throw* is executed as part of an *if* statement, it makes no sense to have an *else* clause:

```
if (strInput == null)
    throw new ArgumentNullException();
else
{
    // Do stuff if exception is not thrown
}
```

You can simply follow the *if* statement containing the *throw* with the other code:

```
if (strInput == null)
    throw new ArgumentNullException();

// Do stuff if exception is not thrown
```

Let's write our own *Parse* method for unsigned integers. Restricting it to unsigned integers simplifies the logic because negative signs won't be allowed. The method will throw the same three exceptions as the normal *Parse* method.

MethodWithThrows.cs

```
//-----------------------------------------------
// MethodWithThrows.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class MethodWithThrows
{
    static void Main()
    {
        uint input;

        Console.Write("Enter an unsigned integer: ");

        try
        {
            input = MyParse(Console.ReadLine());
            Console.WriteLine("You entered {0}", input);
        }
        catch (Exception exc)
        {
            Console.WriteLine(exc.Message);
        }
    }
    static uint MyParse(string str)
    {
        uint result = 0;
        int i = 0;

        // If argument is null, throw an exception.
        if (str == null)
            throw new ArgumentNullException();

        // Get rid of white space.
        str = str.Trim();

        // Check if there's at least one character.
        if (str.Length == 0)
            throw new FormatException();

        // Loop through all the characters in the string.
        while (i < str.Length)
        {
            // If the next character's not a digit, throw exception.
            if (!Char.IsDigit(str, i))
                throw new FormatException();

            // Accumulate the next digit (notice "checked").
            result = checked(10 * result + (uint) str[i] - (uint) '0');

            i++;
        }
        return result;
    }
```

```
}
```

The *MyParse* method trims off any white space using the *Trim* method of the *String* class and then uses a *while* statement to loop through all the characters in the string. If a character passes the *IsDigit* test, the method multiplies result by 10 and adds the new digit converted from Unicode to its numeric value. *MyParse* doesn't explicitly throw an *OverflowException*; instead, it performs the calculation in a *checked* statement to generate the normal *OverflowException*. The *Main* method lets you experiment with *MyParse* and catches any exceptions it may throw.

Although *Parse* certainly provides a good example for exception handling, you actually have an alternative. All the numeric types also support a method named *TryParse*. This method doesn't raise exceptions. Instead, it returns a *bool* indicating if the *string* was successfully parsed. If so, the number is returned as an argument defined using the *out* keyword.

Here's the InputDoubles program converted to use *TryParse*:

InputDoublesWithTryParse.cs

```
//-----------------------------------------------------
// InputDoublesWithTryParse.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;

class InputDoublesWithTryParse
{
    static void Main()
    {
        double num = GetDouble("Enter the base: ");
        double exp = GetDouble("Enter the exponent: ");
        Console.WriteLine("{0} to the power of {1} is {2}",
                    num, exp, Math.Pow(num, exp));
    }
    static double GetDouble(string strPrompt)
    {
        double input;
        Console.Write(strPrompt);

        while (!Double.TryParse(Console.ReadLine(), out input))
        {
            Console.WriteLine();
            Console.WriteLine("You typed an invalid number!");
            Console.WriteLine("Please try again: ");
            Console.WriteLine();
            Console.Write(strPrompt);
        }
        return input;
    }
}
```

Notice that the *while* statement uses the return value from *Double.TryParse* as its Boolean argument. Also notice the second argument to the *TryParse* method includes the *out* keyword.

The *TryParse* method is particularly handy in graphical environments such as Windows Forms. For example, suppose you have a dialog box with an OK button named *btnOK* and a *TextBox* named *txtbox* for the user to enter a floating-point number. You can install a *TextChanged* event handler for the *TextBox* and whenever the text changes, you make the following call:

```
btnOK.Enabled = Double.TryParse(txtbox.Text, out input);
```

Thus, the OK button isn't even enabled until the *TextBox* has proper input.

Chapter 13. Classes, Structures, and Objects

Although the .NET Framework defines many classes and structures—including those that support the basic data types of C# such as *int* and *string*—object-oriented languages such as C# allow you, the programmer, to define your own classes and structures.

Very often, programs of various sorts must deal with calendar dates. It's easy enough to define three integers in your program that you can set to represent a date:

```
int year, month, day;
```

However, if you have a method that must deal with this date in some way, the method would require three parameters. Suppose you want to write a method that determines the number of days between two dates. That method requires six parameters.

For these reasons and others, it is probably much more convenient to deal with a particular date as a single entity rather than three separate numbers. This convenience is a primary impetus behind object-oriented programming.

You might assume that the .NET Framework already defines a class or structure to represent dates, and you would be correct. The *DateTime* structure in the *System* namespace is very important in .NET programming, which is why I devote Chapter 23 to it.

But for this exercise—which begins in this chapter and continues in several subsequent chapters—I'd like to assume that *DateTime* does not exist. Or perhaps, you've decided that you'd rather use an alternative to *DateTime* that you know inside and out.

Before the era of object-oriented programming, many languages allowed programmers to consolidate several variables into entities sometimes referred to as *programmer-defined* data types, or *compound* data types, but very often named *structures*. Here's a C structure to represent a date:

```
struct Date
{
    int year;
    int month;
    int day;
};
```

The keyword *struct* is followed by the name of the structure. The body of the structure is enclosed in curly brackets. This particular structure contains three members, all of which are fields, and all of which are of type *int*.

You can define a *Date* structure the same way in C#, but it wouldn't be very useful. If a structure contains only fields, the fields need to be accessible from outside the structure, and for that reason they must be preceded by the *public* keyword:

```
struct Date
{
    public int year;
    public int month;
    public int day;
}
```

The keyword *public* is known as an *access modifier* and allows the fields to be accessed from code outside the structure. If you want, you can consolidate all three fields in a single declaration:

```
struct Date
{
    public int year, month, day;
}
```

The *public* access modifier applies to all three fields. (However, you can't use the C++ syntax for applying the *public* modifier to multiple fields.)

Also important is the fact that these three fields do not include the *static* keyword, which means these are instance fields rather than static fields. There is no *instance* keyword; members of a class or structure not defined as static are instance by default.

We've had frequent contact with both static methods and instance methods. The expression

```
i.ToString()
```

invokes an instance method that applies to a particular integer—an instance of the Int32 structure. The expression

```
Int32.Parse(str)
```

invokes a static method. Static methods are always prefaced with the class or structure name to which they belong. You don't need an actual integer (otherwise known as an instance of the *Int32* structure) to call *Parse*.

You've also had encounters with static fields, such as the *PI* field in the *Math* class. (Actually *PI* is a constant, but a constant field is also implicitly static.) You refer to a static field by prefacing it with the class name:

```
Math.PI
```

Instance fields are different. You cannot refer to an instance field by prefacing it with the class or structure name. For example, you can't reference the *year* field of the *Date* structure by prefacing it with the name of the structure:

```
Date.year = 1969;    // Won't work!
```

Instead, you must first create an instance of the *Date* structure, which you can do by declaring a variable of type *Date*:

```
Date dateMoonWalk;
```

Often when naming instances of classes or structures, I use a variable name that begins with the lower-case class or structure name, or an abbreviation of that name. You can refer to the variable *dateMoonWalk* is "an instance of the *Date* structure" or "an object of type *Date*."

Once you have declared a *Date* object, you can refer to the fields of that object by prefacing the field with the variable name:

```
dateMoonWalk.year = 1969;
```

Or:

```
Console.WriteLine("The year of the first moon walk was " +
                  dateMoonWalk.year);
```

Here's a complete program that defines the *Date* structure, sets its fields, and then displays the information.

SimpleDateStructureDemo.cs

```
//--------------------------------------------------------
// SimpleDateStructureDemo.cs (c) 2006 by Charles Petzold
//--------------------------------------------------------
using System;

struct Date
{
    public int year;
    public int month;
    public int day;
}

class SimpleDateStructureDemo
{
    static void Main()
    {
        Date dateMoonWalk;

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}/{1}/{2}",
            dateMoonWalk.month, dateMoonWalk.day, dateMoonWalk.year);
    }
}
```

This file contains the definition of the *Date* structure and the definition of a class named *SimpleDateStructureDemo* that contains a *Main* method that refers to the *Date* structure. The order of these two definitions

doesn't matter. The *Date* structure doesn't have to be defined before the *SimpleDateStructureDemo* class.

The structure and the class could also be defined in separate files, but the two files must be part of the same project. If you develop a class or structure that you'd like to reuse in multiple projects, putting that class or structure in its own file is crucial.

You can even define the *Date* structure inside the *SimpleDateStructure-Demo* class (but not inside the *Main* method). The result might look something like this:

```
class SimpleDateStructureDemo
{
    struct Date
    {
        public int year;
        public int month;
        public int day;
    }

    static void Main()
    {
        ...
    }
}
```

However, this *Date* structure would only be accessible from code inside the *SimpleDateStructureDemo* class.

The code in the *Main* method illustrates some unstated "rules" about using the *Date* structure. Both the *month* and *day* fields are one-based rather than zero-based. The *month* value ranges from 1 to 12 for January through December. The *day* value is the familiar day of the month. Of course, this simple *Date* structure has no way of enforcing these rules, but that will be a later enhancement.

I'm also going to set another rule that will come into play later: The *year* field shall refer to years in the common era of the Gregorian calendar. The Gregorian calendar was established by Pope Gregory XIII in 1582 and eventually adopted by much of the rest of the Western world to replace the Julian calendar in effect since the days of Julius Caesar. The Julian calendar had leap years every four years. In the Gregorian calendar, years divisible by 100 are not leap years, except if the year is also divisible by 400.

As you can see, you can declare instances of the *Date* structure as easily as you define instances of the *Int32* structure:

```
Date dateApollo11Launch, dateMoonWalk, dateApollo11SplashDown;
```

However, it is not possible (yet) to define an initial value for the structure right in the declaration statement.

The *SimpleDateStructureDemo* program declares an object of type *Date* like this:

```
Date dateMoonWalk;
```

Alternatively, it could initialize the *Date* structure with a declaration that uses a *new* expression like this:

```
Date dateMoonWalk = new Date();
```

Or, following the simple declaration, you can set *dateMoonWalk* to the *new* expression:

```
dateMoonWalk = new Date();
```

In either case, the *new* expression returns a *Date* structure has all its fields set to zero. In this expression, it has the effect of initializing *dateMoonWalk* to these zero values. The *new* expression essentially "zeroes out" the object. If the structure happened to include fields that were reference types (*string*, for example), those fields would be set to *null*.

Obviously the SimpleDateStructureDemo program doesn't require a *new* expression when defining the *Date* object. The program compiles and runs just fine. That's because the program sets each field of the *Date* object before accessing that field. A field that is not explicitly set—either through a *new* expression or through an assignment statement—is considered to be uninitialized and the C# compiler won't let you access that field.

As your structures start getting larger and more complex, the C# compiler can't always determine whether a particular field has been set or not. It's a good idea to get into the habit of using a *new* expression when defining instances of structures.

In this particular program, if you use a *new* expression but don't set the fields explicitly, the date will be displayed as:

```
0/0/0
```

That date is invalid. It's an invalid month, and invalid day, and an invalid year. There is no year zero. The year before 1 A.D. is 1 B.C.

If at all possible, you should define your structures so that the default value—which is the value the object gets when it's zeroed-out by the *new* expression—is valid and, moreover, represents something akin to zero for the particular object. In theory, the default value for such a *Date* structure should probably be the date January 1 in the year 1, but it doesn't seem quite possible to do that (yet).

You can define an array of *Date* structures using a *new* expression with the data type followed by an array size in square brackets:

```
Date[] dates = new Date[5];
```

This particular *new* expression allocates memory in the heap sufficient to store five *Date* objects. Each *Date* object has three fields of four bytes each, so that's a total of 60 bytes. (Some overhead is also required.)

You can set a particular element of this array like so:

```
dates[3].year = 1969;
dates[3].month = 7;
dates[3].day = 20;
```

And you can display the fields similarly. You index the array with square brackets, and then refer to a field of that element with a period and the field name. In the table of Operator Precedence and Associativity, both array indexing (symbolized by *a[x]* in the table) and the dot operator (*x.y*) have the same precedence and associate left to right.

The following is basically the same program as SimpleDateStructure-Demo except that *Date* is defined as a class rather than a structure.

SimpleDateClassDemo.cs

```
//-------------------------------------------------
// SimpleDateClassDemo.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;

class Date
{
    public int year;
    public int month;
    public int day;
}

class SimpleDateClassProgram
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}/{1}/{2}",
            dateMoonWalk.month, dateMoonWalk.day, dateMoonWalk.year);
    }
}
```

A class is a reference type. When you simply declare a variable using a class like this:

```
Date dateMoonWalk;
```

then *dateMoonWalk* is considered to be uninitialized. It doesn't even equal *null*. No memory has been allocated from the heap. You can't assign anything to the fields because there is no memory to hold the

values. Before you use *dateMoonWalk* at all you must use the *new* operator to create a new instance of the *Date* class. You can do this either in the declaration statement itself

```
Date dateMoonWalk = new Date();
```

or in an assignment statement:

```
dateMoonWalk = new Date();
```

In either case, sufficient memory is allocated from the heap for an instance of the *Date* class. (It's 12 bytes plus overhead.) Because heap memory is automatically set to zero, all the fields of the instance are effectively set to zero.

You can also declare an array of *Date* objects:

```
Date[] dates;
```

At this point the *date*s array is uninitialized. You can allocate memory for the array using a *new* expression by itself

```
dates = new Date[5];
```

or right in the declaration statement:

```
Date[] dates = new Date[5];
```

When *Date* was a structure, the *new* expression allocated memory from the heap sufficient for 5 instances of the structure. When *Date* is a class, memory is allocated from the heap sufficient for 5 *references*. Each of these references is initialized to *null*. What you now have is an array of 5 *null* references.

Before you can use any element of this array, you must use a *new* expression for that element:

```
dates[0] = new Date();
```

Most likely, you'll allocate memory for each element of the array in a *for* loop:

```
for (int i = 0; i < dates.Length; i++)
    dates[i] = new Date();
```

You can't use *foreach* for this job because the array elements are read-only in the body of the *foreach*.

Here's another way to declare, create, and initialize an array of *Date* objects when *Date* is a class:

```
Date[] dates = new Date[5] { new Date(), new Date(), new Date(),
                             new Date(), new Date() };
```

As usual, you can leave out the first *new* expression when you initialize the array in the declaration statement.

Let's review.

Within a method such as *Main* you can declare an object of type *Date* like this:

```
Date dateMoonWalk;
```

Regardless whether *Date* is a class or a structure, space for *dateMoonWalk* is set aside on the stack. If *Date* is a structure, the space on the stack is the size of the structure, which is 12 bytes. If *Date* is a class, the space on the stack is the size of a reference. In each case, the object is uninitialized.

Only if *Date* is a class can you assign *dateMoonWalk* a *null* reference:

```
dateMoonWalk = null;
```

The object is no longer uninitialized, but it doesn't refer to anything.

You can also use the *new* operator with the object:

```
dateMoonWalk = new Date();
```

If *Date* is a structure, then the fields of the structure are all set to zero. The object is now initialized. If *Date* is a class, then memory is allocated from the heap sufficient to store a *Date* object. The heap memory is set to zero, effectively setting the fields of the class to zero. The object is now initialized.

You can also declare an array of *Date* objects:

```
Date[] dates;
```

Space on the stack is set aside for a reference. The *dates* array is a reference regardless whether *Date* is a class or a structure. It is unitialized.

Arrays themselves are always stored in the heap. You use the *new* operator to allocate this heap memory:

```
dates = new Date[27];
```

If *Date* is a structure, then sufficient memory is allocated from the heap to store 27 instances of the *Date* structure, with 12 bytes each. Each of these instances is zeroed-out and considered initialized. If *Date* is a class, memory is allocated to store 27 references. Each element of the array is effectively initialized to *null* and must be allocated individually with a *new* expression.

Instances of structures require less memory and less overhead than instances of classes, and require much less activity when arrays are involved. Structures are suitable for *light-weight* objects, particularly objects that are similar to numbers in some way. Whenever you need a new class or structure that has just a few fields, you should probably make it a structure, especially if you expect to be creating arrays of the objects.

However, there are certainly trade-offs. As you'll see in the chapters ahead, structures have some distinct drawbacks.

Chapter 14. Instance Methods

One common task when working with dates is calculating day-of-year values, which is the number of days from the beginning of the year to a particular day.

One part of this job is determining whether a year is a leap year or not. This static method does that job:

```
static bool IsLeapYear(int year)
{
    return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
}
```

Chapter 11 showed a static *DayOfYear* method that ignored leap years. The method used a static array named *daysCumulative* to simplify the job:

```
static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                181, 212, 243, 273, 304, 334 };

static int DayOfYear(int month, int day)
{
    return daysCumulative[month - 1] + day;
}
```

As we discovered in that chapter, it's better to declare initialized arrays as static fields rather than local variables so they only get initialized once.

The *DayOfYear* method is fairly easy to alter to take account of leap years. All that's necessary is to add 1 when the year is a leap year and the month is March or later:

```
static int DayOfYear(int year, int month, int day)
{
    return daysCumulative[month - 1] + day +
        (month > 2 && IsLeapYear(year) ? 1 : 0);
}
```

Notice that an extra parameter had to be added for the year, and that the method makes use of the *IsLeapYear* method.

However, one of the reasons we originally decided it might be best to treat dates as single entities is to avoid having three parameters to methods like *DayOfYear*. It would be much better having just one parameter of type *Date*:

```
    static int DayOfYear(Date dateParam)
    {
        return daysCumulative[dateParam.month - 1] + dateParam.day +
            (dateParam.month > 2 && IsLeapYear(dateParam.year) ? 1 : 0);
    }
```

Now, rather than referring to the year, month, and day parameters to the method, it refers to *dateParam.year*, *dateParam.month*, and *dateParam.day*. Unfortunately, simplifying the parameter list seems to make the body of the method more complex. Don't worry—it'll shrink back down in size before the end of this chapter.

Let's put all this stuff in a working program. The program is called StructureAndMethodsOne because it is the first in a three-part series that evolves from traditional procedural programming to object-oriented programming. This program declares *Date* as a structure, but nothing in this chapter depends on that. You can change *Date* to a class and all the programs in this chapter will work the same.

StructureAndMethodsOne.cs

```
//-----------------------------------------------------
// StructureAndMethodsOne.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;

struct Date
{
    public int year;
    public int month;
    public int day;
}

class StructureAndMethodsOne
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}/{1}/{2} Day of Year: {3}",
            dateMoonWalk.month, dateMoonWalk.day, dateMoonWalk.year,
            DayOfYear(dateMoonWalk));
    }
    static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };
```

```
    static int DayOfYear(Date dateParam)
    {
        return daysCumulative[dateParam.month - 1] + dateParam.day +
            (dateParam.month > 2 && IsLeapYear(dateParam.year) ? 1 : 0);
    }
}
```

The *Date* structure is the same as the one in the previous chapter. All I've done is add one static field and two static methods to the *StructureAndMethods* class to calculate the day-of-year for a particular *Date* object. The call to *DayOfYear* occurs in the *Console.WriteLine* call in *Main*.

There is nothing really wrong with this program, but there's not much that couldn't also be done in C. One of the objectives of object-oriented programming is to write code that is reusable, and perhaps even accessible from a dynamic link library. If we have a structure named *Date*, then it helps if that structure itself contains methods that involve these dates.

C doesn't allow putting code in a structure, but object-oriented languages like C++ and C# allow classes and structures to contain both code and data. (Of course, you know this already, because classes that contain methods like *Main* can also contain fields.)

StructureAndMethodsTwo.cs

```
//-----------------------------------------------------
// StructureAndMethodsTwo.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;

struct Date
{
    public int year;
    public int month;
    public int day;

    public static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

    public static int DayOfYear(Date dateParam)
    {
        return daysCumulative[dateParam.month - 1] + dateParam.day +
            (dateParam.month > 2 && IsLeapYear(dateParam.year) ? 1 : 0);
    }
}
```

```
class StructureAndMethodsTwo
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}/{1}/{2} Day of Year: {3}",
            dateMoonWalk.month, dateMoonWalk.day, dateMoonWalk.year,
            Date.DayOfYear(dateMoonWalk));
    }
}
```

This is really quite similar to the first version of the program except that stuff has been moved around. I cut and pasted the two static methods and static field into the *Date* structure themselves. Because the *DayOfYear* method needs to be accessed from outside the class I gave it a *public* modifier. I also gave *IsLeapYear* a *public* modifier just in case anyone wants to use that one. But *daysCumulative* I left private under the assumption that this array wouldn't be very important to external classes.

You might notice that *IsLeapYear* has a parameter that is the same name as one of the fields. That's OK. Within the method, *year* refers to the method parameter.

The only real difference in *Main* is how the *DayOfYear* method is called. When it was in the same class as *Main*, it was called like this:

```
DayOfYear(dateMoonWalk)
```

Now it's a static method in the *Date* structure so when called from outside the structure, it must be prefaced with the structure name:

```
Date.DayOfYear(dateMoonWalk)
```

However, *DayOfYear* can still refer to *IsLeapYear* without prefacing it with the structure name because the two methods are in the same structure.

The next enhancement to the program certainly simplifies it. This enhancement changes *DayOfYear* from a static method to an instance method. Moving the static *DayOfYear* method into the *Date* structure was only the preliminary step to this much more important change.

StructureAndMethodsThree.cs

```
//-----------------------------------------------------
// StructureAndMethodsThree.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;
```

```
struct Date
{
    public int year;
    public int month;
    public int day;

    public static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] DaysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

    public int DayOfYear()
    {
        return DaysCumulative[month - 1] + day +
            (month > 2 && IsLeapYear(year) ? 1 : 0);
    }
}

class StructureAndMethodsThree
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}/{1}/{2} Day of Year: {3}",
            dateMoonWalk.month, dateMoonWalk.day, dateMoonWalk.year,
            dateMoonWalk.DayOfYear());
    }
}
```

The static version of *DayOfYear* in the previous program calculated a day-of-year value based on a *Date* parameter to the method. It needed this parameter to reference the three fields of the particular *Date* structure for which it's calculating a day-of-year value.

The instance version of *DayOfYear* has no parameter; instead, it refers directly to the three instance fields of the structure. Notice how much the *DayOfYear* code has been simplified: Rather than *dateParam.month*, it can now simply reference *month*. Any instance method in a class or structure has direct access to the instance fields of that class or structure.

Now look at *Main*. Previously *Main* had to call the static *DayOfYear* method by specifying the *Date* structure in which the method is declared, and passing a *Date* object to the method:

```
Date.DayOfYear(dateMoonWalk)
```

The new version calls the parameterless *DayOfYear* method using the *Date* instance:

```
dateMoonWalk.DayOfYear()
```

The *DayOfYear* method basically needs the same information it did before—a year, a month, and a day. In the static version, it was getting this information through the method parameter. The instance version of the method is always called based on a particular instance of the structure so the method can access the structure instance fields directly.

This code change also brings about a change in perspective. Previously *Main* was asking the *DayOfYear* method in the *Date* structure to calculate a day-of-year value for a particular instance of *Date*. Now the *Main* method is asking the *Date* instance named *dateMoonWalk* to calculate its own day-of-year value.

I've kept *IsLeapYear* a static method just for some variety. Perhaps a static *IsLeapYear* method might be useful if a program wanted to determine if a particular year were a leap year without actually creating an instance of the *Date* structure.

Any instance method in a class or structure can access instance fields in the class or structure; also, any instance method can call any other instance method in the class or structure. Any instance method can also access static fields and call any static method in the class or structure.

However, you can't go the other way. A static method *cannot* access instance fields in the same class or structure, and a static method cannot call instance methods. (Well, actually, a static method *can* call an instance method or access instance fields, but only if that static method has access to an instance of the class—like *DayOfYear* in the *StructureAndMethodsTwo* program. In general, that's not the case.) A static method can't access instance fields or call instance methods because an instance of the class doesn't even have to exist when a static method is called.

The first instance method you encountered in this book was *ToString*. As you discovered, it's possible for a program to pass any object to *Console.WriteLine* and the *ToString* method provides some kind of text representation of the object. You might be curious to try this with the *Date* structure by inserting the statement

```
Console.WriteLine(dateMoonWalk.ToString());
```

or just:

```
Console.WriteLine(dateMoonWalk);
```

In either case, you'll get the string

```
Date
```

which is nothing more nor less than the name of the *Date* structure. The *Date* structure certainly seems to have a *ToString* method, but at the moment it's not doing anything very interesting.

Every class and every structure in the .NET Framework—including those classes and structure you write yourself—has a *ToString* method. The ubiquity of *ToString* is made possible through inheritance, which is one of the primary characteristics of object-oriented programming. When one class inherits (or *derives*) from another class, the new class acquires all the non-private fields and methods of the class it inherits from, and it can add its own fields and methods to the mix. The class that a new class derives from is called the *base* class, and *base* is also a C# keyword.

One of the primary differences between classes and structures involves inheritance. Classes can inherit from other classes, but a structure exists mostly in isolation. A structure cannot explicitly inherit from anything else, and nothing can inherit from a structure.

All classes and structures derive from the grand matriarch of the .NET Framework, *System.Object*. In C#, the keyword *object* is an alias for *System.Object*.

Another important class in the *System* namespace is *System.ValueType*, which inherits directly from *System.Object*. Although structures can't explicitly inherit from anything else, all structures implicitly derive from *System.ValueType*.

The techniques and implications of inheritance will become more evident in the chapters ahead. For now, you should know that the *ToString* method exists in all classes and structures because it's defined in the *System.Object* class like so:

```
public virtual string ToString()
{
    ...
}
```

Actually, just so you won't think the default implementation of *ToString* is a long sophisticated chunk of code, I wouldn't be surprised if it were implemented like this:

```
public virtual string ToString()
{
    return GetType().FullName;
}
```

GetType is another method defined by *System.Object*. It returns an object of type *Type*, and *FullName* is a property of the *Type* class that returns the namespace and name of the type.

At any rate, take note of the *virtual* keyword. This keyword means that any class or structure can provide a custom-made *ToString* method that

supersedes the one declared in *System.Object*. This is known as *overriding* the method, and you do it using the *override* keyword. To provide a custom *ToString* method, you declare it like so:

```
public override string ToString()
{
    ...
}
```

Everything else about the method—the existence of the *public* keyword, the return value of *string*, and the absence of parameters—must be the same.

The *virtual* and *override* keywords are closely related. A *virtual* method in one class can be superseded by an *override* method in a derived class. I'll have more to say about this in Chapter 19.

A custom *ToString* method must return an object of type *string*, and frequently it uses the *String.Format* static method for formatting data for display. Here's a new version of the *Date* structure with such a *ToString* method.

StructureWithToString.cs

```
//-----------------------------------------------------
// StructureWithToString.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;

struct Date
{
    public int year = 1;
    public int month;
    public int day;

    public static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

    public int DayOfYear()
    {
        return daysCumulative[month - 1] + day +
            (month > 2 && IsLeapYear(year) ? 1 : 0);
    }

    static string[] strMonths = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
    public override string ToString()
    {
        return String.Format("{0} {1} {2}", day, strMonths[month - 1], year);
    }
}

class StructureWithToString
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.year = 1969;
        dateMoonWalk.month = 7;
        dateMoonWalk.day = 20;

        Console.WriteLine("Moon walk: {0}, Day of Year: {1}",
            dateMoonWalk, dateMoonWalk.DayOfYear());
    }
}
```

ToString is an instance method, of course, so it can refer directly to the fields of the *Date* structure. It also makes use of a static array of month names, and chooses to format the date in the European fashion.

Now *Main* can display the date just by passing an instance such as *dateMoonWalk* to *Console.WriteLine*. *Console.WriteLine* passes the object to *String.Format*, which calls the object's *ToString* method, which also calls *String.Format*.

But the problem with invalid dates is getting much more critical. Try the two statements:

```
Date dateDefault = new Date();
Console.WriteLine(dateDefault);
```

The *ToString* method of *Date* now throws an exception because it attempts to access *strMonths* with an index of –1. *DayOfYear* has a similar problem. That might suggest to you that *ToString* or *DayOfYear* is at fault and must be fixed. But that's not so. The problem occurs much earlier, when *Date* allows an invalid date to be created in the first place. We must prevent that from happening.

Chapter 15. Constructors

Consider this expression:

```
new Date()
```

What this expression actually does depends on whether *Date* is a structure or a class. If *Date* is a class, the *new* operator allocates memory from the heap sufficient to store an object of type *Date*. This memory must be sufficient for all the instance fields defined in the class. The memory allocation causes these fields to be set to zero, which causes all value-type fields to be set to zero, and all reference-type fields to be set to *null*. The *new* operator returns a reference to the memory in the heap.

If *Date* is a structure, the *new* expression returns a *Date* object with all its fields set to zero, but the *new* expression by itself doesn't really do much of anything. If *dt* is an object of type *Date*, then the expression

```
dt = new Date()
```

effectively sets all the fields of *dt* to zero or *null*.

In either case, this behavior has been causing problems. We want a newly created *Date* object to represent a date of 1/1/1 rather than 0/0/0.

One way to fix this problem is to simply initialize the fields in the declaration statements:

```
class Date
{
    public int year = 1;
    public int month = 1;
    public int day = 1;
    ...
}
```

Now a little more happens during the *new* expression. After the memory for the instance has been allocated from the heap, the three fields are all initialized to 1. It's a valid date.

The only problem is that you can initialize fields only for a class. If *Date* is a structure, then fields are always initialized to zero or *null*, and you can't override that behavior. Try it and you'll get an error message that says you "cannot have instance field initializers in structs."

This prohibition is part of the reduced overhead involved with structures. For an individual structure, it may not seem like much, but it really makes a difference when a program creates an array based on a

structure type. It's much faster to allocate a block of heap memory that's initialized to zero, rather than to initialize each and every element of the array to specific field settings, particularly considering that some of these fields could be *reference* types set to *new* expressions themselves.

Of course, arrays based on classes work much differently. Each element of the array is initialized to *null* and must be created with another *new* expression. Only then is memory allocated for each element and the fields are initialized.

There is another approach to object initialization that's much more generalized than just setting fields. You can initialize fields and perform other arbitrary initialization in special methods known as *constructors*. Constructors are so called because they construct the object. Constructors are methods that are executed automatically when an object is created.

In class or structure definitions, constructors look much like other methods but with two unique characteristics: First, they have the same name as the class itself. In a class or structure named *Date*, a constructor has the name *Date*. Secondly, constructors have no return value.

Here's that *new* expression again:

```
new Date()
```

The right side of that expression looks like a call to a method named *Date* with no arguments. That's a constructor. Here's a skeleton of a *Date* class containing a constructor that initializes the three fields to 1:

```
class Date
{
    int year, month, day;

    public Date()
    {
        year = 1;
        month = 1;
        day = 1;
        ...
    }
    ...
}
```

If *Date* were a regular method, it would have a return type between the *public* keyword and the method name.

A constructor with no parameters is known as a *parameterless* constructor. And here's another difference between classes and structures: You can't declare a parameterless constructor in a structure. (Again, this prohibition exists primarily to speed up array creation

involving structure types.) So now you see two ways of initializing fields in a class, but neither of them works in a structure.

In a class, you can initialize some fields in the declaration statements and some fields in a constructor:

```
class Date
{
    int year;
    int month = 4;
    int day = 1;

    public Date()
    {
        year = 1;
        month = 1;
        ...
    }
    ...
}
```

There's really no difference between these two ways of initializing fields. The constructor that the C# compiler generates (which has the name *.ctor* when you examine it in the IL Disassembler) first contains CIL code that sets the fields in accordance with their declaration statements, and then CIL code for the C# code you explicitly put in body of the constructor. In this odd example, the *month* field ends up as the value 1.

The constructor is required to be *public* if you want to allow code external to the class to create objects. If not, you can define the constructor as *private* or just leave off the *public* modifier.

Besides the parameterless constructor, it's also possible to declare constructors that include parameters. These can be very useful for object initialization. For example, so far we've been creating an initializing a *Date* object like so:

```
Date dateMoonWalk = new Date();
dateMoonWalk.year = 1969;
dateMoonWalk.month = 7;
dateMoonWalk.day = 20;
```

With a parametered constructor, you can conceivably do it like this:

```
Date dateMoonWalk = new Date(1969, 7, 20);
```

Again, the expression on the right looks like a method call, and it essentially is. The constructor has three parameters. Without any consistency checking, it might be defined like this:

```
class Date
{
    int year, month, day;
```

```
    public Date(int yearInit, int monthInit, int dayInit)
    {
        year = yearInit;
        month = monthInit;
        day = dayInit;
    }
    ...
}
```

You can declare a parametered constructor in either a class or a structure.

I gave the three parameters to the constructor slightly different names than the fields just so there's no confusion. You can actually use the same names, and doing that is usually easier than making up slightly different names. To distinguish fields from the parameters, you preface the field names with the keyword *this*:

```
class Date
{
    int year, month, day;

    public Date(int year, int month, int day)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
    ...
}
```

Within instance methods in a class or structure, the keyword *this* refers to the current object. You can actually preface all references to instance fields and instance methods with *this* and a period, but obviously it's not required.

By providing a constructor with multiple parameters in your classes and structures, you're giving programmers who use that class or structure a convenient way to create objects. However, the multi-parameter constructor isn't quite as safe as forcing programmers to set the fields explicitly. If all the parameters are the same type, it's easy to mix them up. (On the other hand, someone using the class or structure might forget to set a field.)

The *C# Language Specification*, §10.10.4 states, "If a class contains no instance constructor declarations, a default instance constructor is automatically provided."

By "default instance constructor" the *C# Language Specification* is referring to the default parameterless constructor. We know this is true because the versions of the *Date* class and structure in earlier chapters contained no instance constructor declarations, but they still seemed to contain a parameterless constructor anyway.

But this statement from the *C# Language Specification* has another profound implication: If you explicitly declare a parametered constructor in your class, then the default parameterless constructor disappears. You'd be able to create a *Date* object like this:

```
Date dt = new Date(2007, 3, 5);
```

But you couldn't do it like this:

```
Date dt = new Date();
```

You'd get the compile error "No overload for method 'Date' takes '0' arguments." If you declare parametered constructors, you also need to explicitly include a parameterless constructor if you want objects to be created using a parameterless constructor. You may not. You may want to prevent objects from being created with a parameterless constructor. It's your choice.

With structures, it doesn't matter if you declare a bunch of constructors with parameters or not. C# continues to provide a public parameterless constructor and there's nothing you can do to make it go away. The implication is simple: You can always create an array of a structure type because structures always have a parameterless constructor.

Parametered constructors are particularly useful when you're declaring an array of initialized objects. Here's some code that is certainly explicit about which fields of which array elements are being set:

```
Date[] dates = new Date[3];

dates[0] = new Date();
dates[0].year = 2007;
dates[0].month = 2;
dates[0].day = 2;

dates[1] = new Date();
dates[1].year = 2007;
dates[1].month = 8;
dates[1].day = 29;

dates[2] = new Date();
dates[2].year = 2007;
dates[2].month = 10;
dates[2].day = 22
```

If *Date* were a structure, the first *new* expression would be required but the others would not. But the real concision comes when *Date* has a parametered constructor. Each element of the array could be set in a single statement:

```
Date[] dates = new Date[3];
dates[0] = new Date(2007, 2, 2);
dates[1] = new Date(2007, 8, 29);
dates[2] = new Date(2007, 10, 22);
```

Or the three elements can be initialized during array creation:

```
Date[] dates = { new Date(2007, 2, 2), new Date(2007, 8, 29),
                 new Date(2007, 10, 22) };
```

What you *cannot* do in C# is initialize an array by just listing the values of the fields as you can in C or C++.

The following program contains a parameterless constructor that initializes the date to 1/1/1, and a constructor with three parameters that performs extensive consistency checking, which now rather dominates the code.

ConsistencyChecking.cs

```
//-------------------------------------------------
// ConsistencyChecking.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;

class Date
{
    public int year;
    public int month;
    public int day;

    // Parameterless constructor
    public Date()
    {
        year = 1;
        month = 1;
        day = 1;
    }

    // Parametered constructor
    public Date(int year, int month, int day)
    {
        if (year < 1)
            throw new ArgumentOutOfRangeException("Year");

        if (month < 1 || month > 12)
            throw new ArgumentOutOfRangeException("Month");

        if (day < 1 || day > 31)
            throw new ArgumentOutOfRangeException("Day");

        if (day == 31 && (month == 4 || month == 6 ||
            month == 9 || month == 11))
            throw new ArgumentOutOfRangeException("Day");

        if (month == 2 && day > 29)
            throw new ArgumentOutOfRangeException("Day");

        if (month == 2 && day == 29 && !IsLeapYear(year))
            throw new ArgumentOutOfRangeException("Day");
```

```
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

    public int DayOfYear()
    {
        return daysCumulative[month - 1] + day +
            (month > 2 && IsLeapYear(year) ? 1 : 0);
    }

    static string[] strMonths = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    public override string ToString()
    {
        return String.Format("{0} {1} {2}", day,
            strMonths[month - 1], year);
    }
}

class ConsistencyChecking
{
    static void Main()
    {
        Date dateDefault = new Date();

        Console.WriteLine("Default Date: {0}", dateDefault);
    }
}
```

A constructor has no return value. If a constructor encounters a problem and can't continue, it has little choice but to raise an exception. For this constructor, I decided that the *ArgumentOutOfRangeException* seemed to best describe the problem. To help in diagnosis, I've provided the exception constructors with arguments indicating whether the year, month, or day was the primary culprit.

Of course, if you're providing a class to other programmers, you'll want to clearly document that a constructor can raise an exception if the input is not valid.

Notice the constructor's use of the *IsLeapYear* method. If *IsLeapYear* were an instance method rather than a static method, the constructor might still be able to use it, but only with some caution. As you'll note, when the constructor calls *IsLeapYear*, it has not yet set the *year* field,

so an instance version of *IsLeapYear* would use the value of *year* when the object was first allocated from the heap, and that would be 0. Be very, very cautious when calling instance methods from constructors.

If a class or structure has multiple constructors, they can make use of each other, but in a very special way. For example, suppose you want to have an additional *Date* constructor with just one parameter:

```
Date dateNewYearsDay = new Date(2008);
```

You might want this constructor to make a *Date* object for January 1 of that year. If you'd prefer not to duplicate some of the code already in the three-parameter constructor, you can define this new constructor like this:

```
public Date(int year) : this(year, 1, 1)
{
}
```

Notice the colon following the parameter list. The colon is followed by something resembling a method call, but with the word *this* with three arguments. This actually refers to the three-parameter constructor called with *month* and *day* values of 1. Before executing the body of the one-parameter constructor (if any), this three-parameter constructor is called.

A call from one constructor to another is called a *constructor initializer*, and it's the only way one constructor can make direct use of another constructor. If you need more flexible ways of sharing code among constructors, you can define some methods (probably static methods) in the class specifically for common initialization tasks that multiple constructors need to perform.

I mentioned earlier that the C# compiler generates code that it inserts into the constructor to initialize instance fields. That generated code is always executed first in any constructor. Next is the call to the constructor initializer, followed by the actual code in the body of the constructor.

If your class contains many constructors, each of the compiled constructors will contain identical CIL code to set the instance fields to their initialized values. To avoid this duplicated code, you might want to *not* initialize the fields in their declaration statements and instead initialize them in the parameterless constructor. The parametered constructors can then execute the code in the parameterless constructor through a constructor initializer.

The opposite approach makes more sense for the *Date* class. The parameterless constructor can use the parametered constructor by including a constructor initializer:

```
public Date() : this(1, 1, 1)
{
}
```

If you don't explicitly provide a constructor initializer, the constructor calls the parameterless constructor in the base class, which is the class that your class derives from. (I'll discuss this process more in Chapter 18.) In effect, every constructor calls one (and only one) constructor initializer before executing the code in the body of the constructor.

A class or structure can contain one *static* constructor, which must have no parameters. The declaration of the static constructor must include the *static* keyword but no other modifier. A static constructor in the *Date* class might look like this:

```
static Date()
{
    ...
}
```

The code in the static constructor is guaranteed to execute before any instance constructor, and before any static member in the class is accessed. Even if you don't explicitly include a static constructor, one is generated in intermediate language to initialize static members, such as the static arrays in the *Date* class. In CIL, the static constructor shows up with the name *.cctor*. (The instance constructors have the names *.ctor*.)

The static constructor is a good place to perform initialization of static fields if a declaration statement isn't quite adequate. I use static constructors in a WPF program named ClockStrip, the source code of which is available for downloading from *www.charlespetzold.com/wpf*. One of these static constructors accesses the Windows registry to obtain time-zone information for all the time zones of the world. Another static constructor assembles a collection of locations and their time zones around the world.

Chapter 16. Concepts of Equality

Before continuing with the *Date* class, it might be useful to explore in more depth the differences between classes (reference types) and structures (value types). As you know, C# supports an equality operator

```
A == B
```

and an inequality operator:

```
A != B
```

The equality operator returns *true* if the two operands are equal; the inequality operator returns *true* if the two operands are not equal. These operators work with numeric types as well as *char*, *bool*, and *string*.

Every class and structure includes a method named *Equals*. This method is defined by the *System.Object* class as a virtual method:

```
public virtual bool Equals(object obj)
```

The method is inherited by all other classes and structures, including those that you define. You can use *Equals* with two strings to determine if they represent the same character string. The comparison is case sensitive. For example:

```
string str = "abc";
bool b1 = str.Equals("abc");
bool b2 = str.Equals("Abc");
```

In this code, *b1* is set to *true* but *b2* is set to *false*. You can also use *Equals* with integers:

```
int i1 = 55, i2 = 55;
bool b3 = i1.Equals(i2);
```

In this code, *b3* is set to *true*.

Because *Equals* is inherited by all other classes and structures, it's helpful to see how it works with classes and structures that you define yourself. Here's a little class named *PointClass* that contains two public fields named *x* and *y*, perhaps to represent a two-dimensional coordinate point.

PointClass.cs

```
//----------------------------------------
// PointClass.cs (c) 2006 by Charles Petzold
//----------------------------------------
class PointClass
{
    public int x, y;
}
```

And here's a structure that defines the same two fields.

PointStruct.cs

```
//-----------------------------------------
// PointStruct.cs (c) 2006 by Charles Petzold
//-----------------------------------------
struct PointStruct
{
    public int x, y;
}
```

The EqualsTest project includes both those files and EqualsText.cs, shown below. (To add additional source code files to a project in Visual Studio, just right click the project name in the solution explorer, and choose Add and then New Item from the menu. Or, pick Add New Item from the Project menu.)

EqualsTest.cs

```
//-----------------------------------------
// EqualsTest.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class EqualsTest
{
    static void Main()
    {
        PointStruct ps1 = new PointStruct();
        ps1.x = ps1.y = 55;

        PointStruct ps2 = new PointStruct();
        ps2.x = ps2.y = 55;

        PointClass pc1 = new PointClass();
        pc1.x = pc1.y = 55;

        PointClass pc2 = new PointClass();
        pc2.x = pc2.y = 55;

        Console.WriteLine("ps1.Equals(ps2) results in " + ps1.Equals(ps2));
        Console.WriteLine("ps1.Equals(pc1) results in " + ps1.Equals(pc1));
        Console.WriteLine("pc1.Equals(pc2) results in " + pc1.Equals(pc2));
        Console.WriteLine("pc1 == pc2 results in " + (pc1 == pc2));
        // Console.WriteLine("ps1 == ps2 results in " + (ps1 == ps2));
    }
}
```

The *Main* method creates two *PointStruct* objects named *ps1* and *ps2*, and two *PointClass* objects named *pc1* and *pc2*, and sets all the fields to 55. It then compares them using the *Equals* method and (for the two classes only) the equality operator. Here's what the program displays:

```
ps1.Equals(ps2) results in True
ps1.Equals(pc1) results in False
pc1.Equals(pc2) results in False
pc1 == pc2 results in False
```

The first *Equals* call compares the two structures, and this call returns *true*. That should make us happy that two structures with their fields set to the same value are considered to be equal.

The second *Equals* call compares one of the structures with one of the classes, and it returns *false*. Maybe it just doesn't make sense to compare two objects of different types.

The third *Equals* call compares the two instances of the *PointClass* class, and here is revealed something rather disturbing. *Equals* returns *false*, even though the two objects appear to be identical. Moreover, comparing the two objects with the equality operator also returns *false*. Why is this?

A class is a reference type. What are *pc1* and *pc2* really? They are references to memory blocks that have been allocated in the heap. The two objects might contain identical information but they are separate and distinct. The references are not equal. For that reason, the *Equals* method in *Object* (which *PointClass* inherits) returns *false*.

The *ValueType* class (which is the implicit base class of all structures) overrides the *Equals* method, so that's the one that applies to all structures. *ValueType* implements its own concept of *Equals* to perform a different type of comparison that involves the actual fields of the object. If the two objects are instances of the same structure, and all the fields are equal, then *Equals* returns *true*. This is called a *bitwise* equality test.

You can use the equality operator with classes, and it returns the same value as the *Equals* method. The default equality operator implements reference equality.

The equality and inequality operators are not implemented for structure types. If you want a structure to implement the equality and inequality operators, you'll have to define them yourself, as I demonstrate in Chapter 20.

This different notion of equality also implies a distinction between reference types and value types that involves their application. Structures are best suited for creating objects that are somewhat numeric-like. Two coordinate points with the same *x* and *y* values should certainly be treated as equal.

However, suppose you're dealing with objects that represent people. You define fields to store the person's name, birth date, height, and weight. If two people have the same name, birth date, height, and weight, are they the same person? Not necessarily, and perhaps the fact that these two people are represented by two different objects tells you that these are

actually two separate people. (However, if the two objects had the same value in the *SocialSecurityNumber* field, then they probably do represent the same person and should probably be treated as equal.) Similarly, if two graphical buttons have the same text, and the same size, and the same location on the screen, are they equal? Are they the same button? No. If they're two different objects, then they're two different buttons, and one is sitting on top of the other.

Whenever you create a class or structure, you should give some thought to the concept of equality. If your concept doesn't match the default implementation, then you can override the *Equals* method. (I'll show you how shortly.) You can also define equality and inequality operators as I'll demonstrate in Chapter 20.

Consider the *String* class. Because *String* is a class, the *Equals* method and equality operators would normally implement reference equality. But *String* overrides this method and the equality operators so they perform comparisons of the character strings rather than comparisons of the references. This type of comparison is more valuable to us.

A related difference between classes and structures involves assignment. Suppose you define two integers like so:

```
int i1 = 22, i2 = 33;
```

First you set *i1* equal to *i2*, and then you set *i2* to something else:

```
i1 = i2;
i2 = 55;
```

What does *i1* equal? Obviously 33, the value it obtained when it was assigned the original value of *i2*. But sometimes assignment isn't quite as intuitive, because sometimes assignment involves references.

Here's another project named AssignmentTest. The AssignmentTest.cs file is shown below, but this project also makes use of the PointClass.cs and PointStruct.cs files from the EqualityTest project.

If you're typing these files in yourself, you probably don't want multiple copies of the files hanging around. Instead, you want the AssignmentTest project to have links to the existing files. In Visual Studio, you can right-click the project name in the solution explorer, and choose Add and Existing Item from the menu. Or pick Add Existing Item from the Project menu. Navigate to the EqualityTest directory, pick the files you want links to, click the arrow next to the OK button, and pick Add As Link.

AssignmentTest.cs

```
//-----------------------------------------------
// AssignmentTest.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
```

```
class AssignmentTest
{
    static void Main()
    {
        PointStruct ps1 = new PointStruct();
        ps1.x = ps1.y = 22;

        PointStruct ps2 = new PointStruct();
        ps2.x = ps2.y = 33;

        ps1 = ps2;
        ps2.x = ps2.y = 55;

        Console.WriteLine("ps1 is ({0}, {1})", ps1.x, ps1.y);
        Console.WriteLine("ps2 is ({0}, {1})", ps2.x, ps2.y);
        Console.WriteLine("ps1.Equals(ps2) results in " + ps1.Equals(ps2));

        PointClass pc1 = new PointClass();
        pc1.x = pc1.y = 22;

        PointClass pc2 = new PointClass();
        pc2.x = pc2.y = 33;

        pc1 = pc2;
        pc2.x = pc2.y = 55;

        Console.WriteLine("pc1 is ({0}, {1})", pc1.x, pc1.y);
        Console.WriteLine("pc2 is ({0}, {1})", pc2.x, pc2.y);
        Console.WriteLine("pc1.Equals(pc2) results in " + pc1.Equals(pc2));
        Console.WriteLine("pc1 == pc2 results in " + (pc1 == pc2));
    }
}
```

The first half of *Main* roughly parallels the example I just showed with integers, but with two structures named *ps1* and *ps2*. The two fields of *ps1* are first both assigned 22, and the two fields of *ps2* get 33. Then *ps1* is set to *ps2*:

```
ps1 = ps2;
```

The two fields of *ps2* are then assigned 55. What does *ps1* equal? Here's what the program reports:

```
ps1 is (33, 33)
ps2 is (55, 55)
ps1.Equals(ps2) results in False
```

The two fields of *ps1* are both the values obtained when *ps1* was assigned from *ps2*, rather than the values later set to *ps2*. The code works the same as with the integers.

The second half of *Main* contains parallel code but using *PointClass* rather than *PointStruct*. Two objects named *pc1* and *pc2* are created and assigned values, and *pc1* is set to *pc2*:

```
pc1 = pc2;
```

The program then assigns 55 to the two fields of pc2, and displays the results:

```
pc1 is (55, 55)
pc2 is (55, 55)
pc1.Equals(pc2) results in True
pc1 == pc2 results in True
```

What happened here? Why does *pc1* have the same values later assigned to *pc2*?

A class is a reference type. The *pc1* and *pc2* variables are references to memory blocks allocated in the heap. Following the assignment statement

```
pc1 = pc2;
```

both *pc1* and *pc2* store the same reference, and hence refer to the same memory block. Whatever you do to the fields of *pc1* also affects the fields of *pc2*; likewise, any change to the fields of *pc2* also affects the fields of *pc1*. The *pc1* reference equals the *pc2* reference, as the program demonstrates at the end by using the *Equals* method and equality operator.

Because *PointStruct* is a structure, the statement

```
PointStruct ps1 = new PointStruct();
```

does not result in any memory allocations from the heap. The *ps1* variable is stored on the stack and the *new* operator simply sets all its fields equal to 0 or *null*. In contrast, the statement

```
PointClass pc1 = new PointClass();
```

causes a memory allocation from the heap. Similarly,

```
PointClass pc2 = new PointClass();
```

requires another memory allocation. Following the assignment statement

```
pc1 = pc2;
```

both variables are the same value and refer to the second block of memory allocated from the heap.

What happens to the first block of memory? It seems to be orphaned, and in this simple program, it is. All the references to that first block of memory are now gone. There is no way for the program to get back that reference. The block therefore becomes eligible for garbage collection. The system can free the memory block if necessary to obtain more memory space.

Another related issue involves objects passed to method calls. The following program also has links to the PointClass.cs and PointStruct.cs files and defines methods that change the values of the class and structure fields.

MethodCallTest.cs

```
//-----------------------------------------------
// MethodCallTest.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class MethodCallTest
{
    static void Main()
    {
        PointStruct ps = new PointStruct();
        ps.x = ps.y = 22;

        Console.WriteLine("Before method: ps is ({0}, {1})", ps.x, ps.y);
        ChangeStructure(ps);
        Console.WriteLine("After method:  ps is ({0}, {1})", ps.x, ps.y);

        PointClass pc = new PointClass();
        pc.x = pc.y = 22;

        Console.WriteLine("Before method: pc is ({0}, {1})", pc.x, pc.y);
        ChangeClass(pc);
        Console.WriteLine("After method:  pc is ({0}, {1})", pc.x, pc.y);
    }
    static void ChangeStructure(PointStruct ps)
    {
        ps.x = ps.y = 33;
    }
    static void ChangeClass(PointClass pc)
    {
        pc.x = pc.y = 33;
    }
}
```

When exploring arrays, you discovered that methods can change elements of an array passed to the method. Methods can also change fields of classes but not fields of structures. The program displays the results:

```
Before method: ps is (22, 22)
After method:  ps is (22, 22)
Before method: pc is (22, 22)
After method:  pc is (33, 33)
```

When a program passes an object to a method in preparation for a method call, what happens depends on whether the object is based on a class or a structure. If the object is a value type, a bitwise copy is made for use by the method. If the object is a reference type, the reference itself is copied for use by the method. The method can use this reference to change any field of the class. These changes affect the original object.

In both cases, the method is always working with a copy. It's a bitwise copy for a value type, and a copy of the reference for a reference type.

The *ChangeClass* method in the MethodCallTest program might even conclude by setting the parameter to *null*:

```
static void ChangeClass(PointClass pc)
{
    pc.x = pc.y = 33;
    pc = null;
}
```

The program will work the same. The original reference stored in *Main* is not affected because the method gets a copy of that reference.

Of course, you can write a method that changes the fields of a structure by defining the parameter with the *ref* or *out* keyword. This causes the structure to be passed by reference rather than value. It actually improves efficiency for large value types because the entire structure doesn't need to be copied to the stack, only a reference.

Assignment, method calls, and tests for equality all work a little differently for objects based on structures or classes. Underlying these differences is an important fact regarding the *new* operator.

For classes, a call to the *new* operator is required to create a new instance of the class. The *new* operator initiates some very serious activity. The *new* operator allocates memory from the heap to store the object and calls one of the class's constructors. This is not something you want happening arbitrarily, and usually it does not. For example:

```
PointClass pc1 = new PointClass();
PointClass pc2 = pc1;
```

There's only one *new* expression in this code, so only one instance of *PointClass* is created. Both *pc1* and *pc2* refer to that single instance. The following code is much different because it creates two distinct objects:

```
PointClass pc1 = new PointClass();
PointClass pc2 = new PointClass();
```

Similarly, a *new* operation doesn't occur when an object is passed to a method. The method is working with the same object passed to it.

This whole issue becomes crucial when you start working with classes with constructors that do more than just initialize a few fields. Some classes have constructors that open disk files in preparation for reading or writing, for example. You don't want the same file reopened if you happen to pass the object to a method. When working with user interface objects such as buttons or dialog boxes, you certainly don't want new objects being created when you pass them to a method.

Creating a new instance of a class is often serious business, and that's why it doesn't usually happen without you, the programmer, knowing about it.

Creating a new instance of a structure is much less serious. This code involving a structure

```
PointStruct ps1 = new PointStruct();
PointStruct ps2 = ps1;
```

is equivalent to code that has two *new* operators:

```
PointStruct ps1 = new PointStruct();
PointStruct ps2 = new PointStruct();
```

These two little blocks of code are equivalent because the parameterless constructor of a structure only initializes all the structure fields to zero or *null*. And it can *never* do anything else. You aren't allowed to initialize the fields of a structure to non-zero values or write your own parameterless constructor. That privilege is reserved for classes.

Chapter 17. Fields and Properties

Classes and structures have several types of members, most notably fields, methods, constructors, and properties. You've already encountered several properties in this book, but I haven't yet emphasized how important they've become in the vast scheme of .NET. When C# first emerged from Microsoft, properties seemed like merely a syntactical convenience for programmers. In recent years, properties have evolved into an extremely important feature of the .NET Framework.

Properties have become vitally important mostly because XML has become important. In recent programming interfaces such as the Windows Presentation Foundation (WPF), programmers can even use an XML-based Extensible Application Markup Language (XAML) is defining layout of windows and controls.

Here's a little WPF code:

```
Button btn = new Button();
btn.Foreground = Brushes.LightSeaGreen;
btn.FontSize = 32;
btn.Content = "Hello, WPF!";
```

Notice the use of a parameterless constructor followed by code that sets three properties of the *Button* class. Here's the equivalent in XAML:

```
<Button Foreground="LightSeaGreen" FontSize="32"
        Content="Hello, WPF!" />
```

This translation between code and XAML is so easy and straightforward primarily because *Button* has a parameterless constructor and defines properties to set the color, font size, and content. Imagine the translation mess if *Button* had only a parametered constructor, and if it had various methods rather than properties to set its characteristics.

Let's put some properties in the *Date* class. We definitely need *something* to fix it, because despite the consistency checks implemented in the three-parameter constructor of the *Date* class in Chapter 15, the class is still not safe from invalid dates. A program using the class can create a valid date using one of the constructors and then make the date invalid simply by setting a field or two:

```
Date dt = new Date(2007, 2, 2);
dt.day = 31;
```

This is possible because the fields are public. If you don't want the fields in a class to be changed by programs using the class, you should make them private:

```
private int year;
private int month;
private int day;
```

Because *private* is the default, you can simply remove the access modifier entirely:

```
int year;
int month;
int day;
```

That solves the invalid date problem, but at a severe cost. Code that happens to encounter a *Date* object can't even determine what date is represented by the object! (Well, that's not entirely true. Some code could call *ToString* and then examine the string that's returned to figure out the date.) A better solution is to make the fields *public* but also to mark them as read-only:

```
public readonly int year;
public readonly int month;
public readonly int day;
```

The *readonly* modifier can only be used on fields. Unless you're working with a structure, you can initialize a read-only field in the declaration:

```
public readonly int year = 1;
```

You can also set a read-only field in a constructor:

```
class Date
{
    public readonly int year, month day;

    public Date(int year, int month, int day)
    {
        this.year = year;
        ...
    }
    ...
}
```

But after the constructor finishes, a read-only field is fixed and cannot be changed.

Using *readonly* on the fields essentially makes the *Date* object immutable. You set the value at creation, and then you're stuck with it. That's not necessarily a bad thing. An immutable object may be exactly what you need in some cases. The *DateTime* structure in the .NET Framework is immutable.

By now you've seen several modifiers you can use with fields and methods. These modifiers always appear before the type of the field or the return type of the method, but they can be in any order among themselves.

The access modifiers you've seen are *public* and *private*. (There are actually three more access modifiers—*protected*, *internal*, and *internal protected*.) These five modifiers are mutually exclusive and indicate whether a field or method is accessible from outside the class (or structure) or whether its use is restricted to inside the class.

The *static* modifier indicates that the field or method is associated with the class (or structure) rather than an instance of the class. You refer to a static field or method by prefacing it with the class or structure name. You refer to an instance field or method by prefacing it with an object name. A static method cannot refer to an instance field in the same class or structure, and cannot call an instance method.

You can use *const* to create a constant. It's not quite correct to say that a constant is a field. (The *C# Language Specification* discusses them in two separate sections, §10.3 and §10.4. However, in the .NET Framework class library documentation, constants are listed under the category of Fields.) As with local constants (that is, constants declared inside of methods), the value of a constant must be set in the declaration, and it must be available at compile time. The value of a constant cannot be changed by code. You cannot use the *static* keyword with constants, but constants are inherently static and shared among all instances of the class. If a constant field is *public*, you can refer to it from outside the class by prefacing it with the class name. *Math.PI* is a constant.

The *readonly* modifier is valid only with fields. The value of a read-only field must be set in the declaration or by a constructor. After the conclusion of the constructor, a read-only field cannot be modified.

The *static*, *const*, and *readonly* modifiers are somewhat related: A constant has only a single value regardless of the instance, so a constant is inherently static. A *readonly* field is generally an instance field, but after a constructor finishes execution, it becomes similar to a constant in that it cannot be changed.

There are times when you want to declare a constant, but the value is not available at compile time. This is the case if you're trying to set a constant using a *new* expression:

```
const Date dateSputnik = new Date(1957, 10, 4);     // Won't work!
```

Expression using *new* can be executed only at runtime. The *C# Language Specification*, §10.4.2.1 has the solution: Use *static readonly* rather than *const*:

```
static readonly Date dateSputnik = new Date(1957, 10, 4);
```

However, this guarantees only that *dateSputnik* cannot be set to another *Date* object; it does not guarantee that the Date object referenced by *dateSputnik* is immune from being changed itself, unless it defined in such a way to be immutable.

You may be happy with immutable *Date* objects with *readonly* fields that cannot be changed after the constructor terminates. But you may want a *Date* object that can later change. If so, let's pursue a different approach. Let's keep the fields private, but let's provide methods in the class that can access and change the private fields. The methods that change the private fields can also perform consistency checking and prevent the creation of invalid dates.

Traditionally, such methods begin with the words *Set* and *Get*. Here's a public *SetYear* method to change the private *year* field:

```
public void SetYear(int year)
{
    if (year < 1)
        throw new ArgumentOutOfRangeException("Year");

    this.year = year;
}
```

The method is public. The field is private. There's actually a problem with this consistency check that I'll discuss shortly. For now, I'm sure you get the general idea: The *Set* method can contain code to safeguard against invalid dates. The *GetYear* method is quite simple:

```
public int GetYear()
{
    return year;
}
```

The two methods are somewhat symmetrical. The *SetYear* method has a single *int* argument and no return value, indicated by the keyword *void*. The *GetYear* method has no argument but returns *int*. Likewise, you could also write *SetMonth*, *GetMonth*, *SetDay*, and *GetDay* methods.

Now you can either create a *Date* object using the three-parameter constructor, or you can create a *Date* object using the parameterless constructor and then call the *Set* methods:

```
Date dateSputnik = new Date();
dateSputnik.SetYear(1957);
dateSputnik.SetMonth(10);
dateSputnik.SetDay(4);
```

Likewise, you could access the year, month, and day using *GetYear*, *GetMonth*, and *GetDay* methods.

But like I said, that's the *traditional* approach. When programming in C# you can instead use properties, which are a little cleaner, a little easier, and a little prettier.

Like methods and fields and constructors, properties are members of classes and structures. We've already encountered some properties of the basic data types. The *Length* property of the *String* class indicates the number of characters in the string. The *Array* class defines both a *Length*

property and a *Rank* property. All these properties happen to be read-only, but properties that you declare in your classes and structures don't have to be.

In use, properties look exactly like fields. Here's how you might define a *Date* object using properties named *Year*, *Month*, and *Day*:

```
Date dateSputnik = new Date();
dateSputnik.Year = 1957;
dateSputnik.Month = 10;
dateSputnik.Day = 4;
```

Property names are often capitalized. Just as you can set properties as if they were fields, you can access properties as if they were fields:

```
Console.WriteLine("Sputnik was launched in " + dateSputnik.Year);
```

Both pieces of code are more attractive and readable than the equivalent code using *Set* and *Get* methods.

And yet, properties contain code, so they are just as powerful as *Set* and *Get* methods. For this reason, properties are often called "smart" fields because they can add a little code (for consistency and validity checks) to setting and accessing fields. Often a public property is associated with a private field (sometimes called a "backing field") but that's not a requirement.

Property declarations have a special syntax in C# that distinguishes them from both methods and fields. Here's a declaration of a *Year* property that is functionally equivalent to the *SetYear* and *GetYear* methods I showed earlier:

```
public int Year
{
    set
    {
        if (value < 1)
            throw new ArgumentOutOfRangeException("Year");

        year = value;
    }
    get
    {
        return year;
    }
}
```

The property declaration begins with an optional access modifier. Properties are very often *public*. Properties can also include the *static* modifier or any other modifier used with methods. The type of the property follows. Here it's *int*. That's both the type of the argument to the earlier *SetYear* method and the return type of the *GetYear* method. That symmetry is how both methods can be combined into a single property.

The name of the property is *Year*. A left curly bracket follows. That's how the compiler knows it's not a method or a field. If *Year* were a method, it would be followed by a left parenthesis. If *Year* were a field, it would be followed by a comma, equal sign, or semicolon.

Within the outer set of curly brackets are one or two sections (called *accessors*) that begin with the words *get* and *set*. Often both sections are present. If only a *get* accessor is present, the property is read-only. (You might want to say "get only" or "gettable" instead so everyone knows you're talking about properties.) A property can have only a *set* accessor, but that's rather rare. The words *get* and *set* are not considered C# keywords because you can use the words as variable names. They have a special meaning only in this particular place in the property declaration.

Within the body of the property declaration, *set* and *get* are both followed by another set of curly brackets. In the *set* accessor, the special word *value* refers to the value being set to the property; the *get* accessor must have a *return* statement to return a value.

Properties are not actually part of the Common Language Specification (CLS) nor are they implemented in intermediate language. When you declare a property named *Year*, for example, the C# compiler fabricates methods named *set_Year* and *get_Year* that contain the property code. If you use a language that does not support properties (such as C++), you'll have to refer to properties using these method names. You can't have method names in your C# class that duplicate the names that the compiler generates. (See the *C# Language Specification*, §10.2.7.1.) If you declare a *Year* property, you can't also declare a method named *get_Year*.

Programmers experienced in .NET have pretty much come to the conclusion that instance fields should always be private, and that public access to these private fields should be through public properties that guard against invalid values. Public properties always have capitalized names; private fields often have names that begin with lowercase letters, or perhaps underscores.

I haven't shown code yet for the *Month* and *Day* properties, partially because I'm not happy with the code I showed you for the *Year* property. I mentioned it had a problem. Suppose you create a *Date* object like so:

```
Date dt = new Date(2008, 2, 29);
```

That's a valid date because 2008 is a leap year. But suppose the *Year* property is now set to something else:

```
dt.Year = 2007;
```

That's now an invalid date, but the *Year* property as written above didn't catch the problem. Here's a better *Year* property:

```
public int Year
{
    set
    {
        if (value < 1 || (!IsLeapYear(value) && Month == 2 &&
                                                Day == 29))
            throw new ArgumentOutOfRangeException("Year");

        year = value;
    }
    get
    {
        return year;
    }
}
```

Even with this enhanced *Year* property, some consistency-checking problems will persist. Here's that leap-day date again:

```
Date dt = new Date(2008, 2, 29);
```

Now the program sets three properties to something else:

```
dt.Year = 2007;
dt.Month = 3;
dt.Day = 1;
```

The intention is clear from the code that the resultant date is valid, and yet as soon as the *Year* property is set, an exception will be raised. Rearrange the order of the statements and nobody will complain:

```
dt.Month = 3;
dt.Day = 1;
dt.Year = 2007;
```

This problem is not solvable. If you want to provide public properties that allow a program to set a new date, and you want to prevent invalid dates, some sequences of code won't work in the order they're written. But the worst that can be said is that the code is overprotective.

The following file is devoted solely to a class named *Date*. This is the final version of *Date* so it's in its own file (named Date.cs) and can be used by other programs. The class contains *Year*, *Month*, and *Day* properties. I like to arrange my classes so they begin with private fields, and the public properties follow. These are followed by constructors and then methods. Among these methods is a static method named *IsConsistent* that all three properties use to prevent the occurrence of invalid dates.

Date.cs

```
//-------------------------------------
// Date.cs (c) 2006 by Charles Petzold
//-------------------------------------
using System;
```

```
class Date
{
    // Private fields
    int year = 1;
    int month = 1;
    int day = 1;

    // Public properties
    public int Year
    {
        set
        {
            if (!IsConsistent(value, Month, Day))
                throw new ArgumentOutOfRangeException("Year");

            year = value;
        }
        get
        {
            return year;
        }
    }

    public int Month
    {
        set
        {
            if (!IsConsistent(Year, value, Day))
                throw new ArgumentOutOfRangeException("Month = " + value);

            month = value;
        }
        get
        {
            return month;
        }
    }

    public int Day
    {
        set
        {
            if (!IsConsistent(Year, Month, value))
                throw new ArgumentOutOfRangeException("Day");

            day = value;
        }
        get
        {
            return day;
        }
    }
```

```
    // Parameterless constructor
    public Date()
    {
    }

    // Parametered constructor
    public Date(int year, int month, int day)
    {
        Year = year;
        Month = month;
        Day = day;
    }

    // Private method used by the properties
    static bool IsConsistent(int year, int month, int day)
    {
        if (year < 1)
            return false;

        if (month < 1 || month > 12)
            return false;

        if (day < 1 || day > 31)
            return false;

        if (day == 31 && (month == 4 || month == 6 ||
                          month == 9 || month == 11))
            return false;

        if (month == 2 && day > 29)
            return false;

        if (month == 2 && day == 29 && !IsLeapYear(year))
            return false;

        return true;
    }

    // Public properties
    public static bool IsLeapYear(int year)
    {
        return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
    }

    static int[] daysCumulative = { 0, 31, 59, 90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

    public int DayOfYear()
    {
        return daysCumulative[Month - 1] + Day +
            (Month > 2 && IsLeapYear(Year) ? 1 : 0);
    }

    static string[] strMonths = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
    public override string ToString()
    {
        return String.Format("{0} {1} {2}", Day,
            strMonths[Month - 1], Year);
    }
}
```

Notice that the three-parameter constructor sets the properties rather than the fields. Even the *set* accessors of the properties refer to the other properties rather than access the fields. This is not a requirement. Methods in a class can use either the class properties or the fields themselves. But you'll see shortly why I like to structure my classes so that field accesses are kept to a minimum.

Here's a simple program to test out the *Date* class:

PropertyTest.cs

```
//-------------------------------------------
// PropertyTest.cs (c) 2006 by Charles Petzold
//-------------------------------------------
using System;

class PropertyTest
{
    static void Main()
    {
        Date dateMoonWalk = new Date();

        dateMoonWalk.Year = 1969;
        dateMoonWalk.Month = 7;
        dateMoonWalk.Day = 20;

        Console.WriteLine("Moon walk: {0}, Day of Year: {1}",
            dateMoonWalk, dateMoonWalk.DayOfYear());
    }
}
```

Both Date.cs and PropertyTest.cs are part of the PropertyTest project. You can experiment with other dates to make sure the class is working as it should. Of course, in a real program, any code that has the potential of causing *Date* to raise an exception should be put in a *try* block.

Sometimes it's not clear whether a particular chunk of code should be a method or a property. Properties are often considered *characteristics* of an object, whereas methods often perform *actions*. If the object itself is a noun, a property is an adjective and a method is a verb. If you can associate the words *get* and *set* with a method, you probably want to make it a property. I'd be inclined to make *DayOfYear* a read-only property, for example. The only real rule is this: If it needs a parameter to get the value or an extra parameter to set the value, it's got to be a method.

When we first embarked on the idea of encapsulating a date, we explored implementing it as a class or a structure. It soon became clear that using a structure had distinct problems. When an object is created from a structure using a parameterless constructor, there is no way to initialize the fields to anything but 0 or *null*.

Let's ask ourselves the question again: How can we implement *Date* as a structure and prevent an invalid date when the parameterless constructor is used to create the object?

Big hint: The *get* accessor of a property doesn't necessarily have to return simply the value of a field.

The *get* accessor *could* return the value of the field *plus one*. In other words, the private *year*, *month*, and *day* fields could be zero-based, but the public *Year*, *Month*, and *Day* properties could be one-based. Here's a revised *Year* property that uses this technique:

```
public int Year
{
    set
    {
        if (!IsConsistent(value, Month, Day))
            throw new ArgumentOutOfRangeException("Year");

        year = value - 1;
    }
    get
    {
        return year + 1;
    }
}
```

The only difference is that the *set* accessor sets the *year* field to *value* minus one, and the *get* accessor returns the *year* field plus one. The default *year* field is 0 but the *Year* property returns 1. Properties provide a type of buffer around fields so you can make the internals of a class or structure different from how it looks from outside.

Suppose you wanted to make the days of the month available as a public property. You could define a public static property of type *string* array and simply return the entire array of month names:

```
public static string[] MonthNames
{
    get
    {
        return strMonths;
    }
}
```

Then, if a program wanted the abbreviated name for January, for example, the expression would be *Date.MonthNames[0]*. It seems as if the property returns the whole array, but it's really only returning a

reference to the array, which is then indexed by the code accessing the property. (The *DateTimeFormatInfo* class in the *System.Globalization* namespace has a *MonthNames* property that returns a string array but the month names are specific to a particular culture and language.)

It's also possible for a class to have a special member called an *indexer* that is declared somewhat like a property. An indexer is intended for classes or structure that store collections of items, so it's not quite appropriate for the *Date* class.

But just suppose that you have a particular application that for a particular *Date* object named *dt*, for example, it is sometimes convenient to refer to the property *dt.Year* as *dt[0]*, and refer to the property *dt.Month* as *dt[1]*, and the property *dt.Day* as *dt[2]*. You actually want to index the object as if it were an array.

Here's how such an indexer would look:

```
public int this[int i]
{
    get
    {
        switch (i)
        {
        case 0: return Year;
        case 1: return Month;
        case 2: return Day;
        default: throw ArgumentOutOfRangeException("index = " + i);
        }
    }
}
```

Notice that the declaration of the indexer begins by resembling a property named *this*, which is then followed by square brackets containing the value of the indexer. As implemented here, this is a read-only indexer, but you could come up with *set* logic as well if you wanted.

You can have multiple indexers if the index is a different type. It doesn't have to be an integer. This one lets you specify a text index:

```
public int this[string str]
{
    get
    {
        switch (str.ToLower())
        {
        case "year": return Year;
        case "month": return Month;
        case "day": return Day;
        default: throw ArgumentOutOfRangeException("index = " + i);
        }
    }
}
```

With an indexer like this, if *dt* is an object of type *Date*, then *dt["MONTH"]* obtains the month.

In the .NET documentation, indexers often show up as the property name *Item*. (In the *String* class, the name is *Chars*.) When programming in C#, you never use that name to refer to indexers. You simply index the object. As with properties, C# fabricates indexers by creating methods named *get_Item* and *set_Item*.

Chapter 18. Inheritance

No class is an island. All classes are related to each other in some way. All classes automatically have methods named *ToString* and *Equals* because these two methods are defined in the class *System.Object*, a class also referred to by the C# keyword *object*. All other classes contain the *ToString* and *Equals* methods because all classes ultimately *derive* (or *inherit*) from *System.Object*.

Inheritance is one of the primary features of object-oriented programming. When a new class derives from an existing class (or *subclasses* an existing class), it inherits all the non-private methods, properties, and fields declared in the existing class. The new class can then extend the existing class by adding to or replacing those methods, properties, and fields. The process is cumulative. A class contains all the non-private methods, properties, and fields declared in itself and all its ancestor classes going back to *System.Object*.

It's all about reusing code. Inheritance provides a structured way to reuse code that's already been written, but inheritance also provides a way to alter or enhance the code in ways that make it more useful or convenient.

The ability to inherit is one of the big differences between classes and structures. Structures do not allow inheritance. All structures implicitly derive from *System.ValueType*, which itself derives from *System.Object*. But you can't define a class that inherits from a structure, or define a structure that inherits from another structure. Once you define a structure, that's the end of the line.

There are many reasons why you'd want to use inheritance. Suppose you have a class that's fine for most of your requirements, but it needs just a few little changes. Even if you have access to the source code, you might not want to change the original class. Maybe the class is working well in some other application, and you wisely respect the "if it ain't broke don't fix it" rule. Or perhaps you don't have access to the code. Maybe the class is available only in compiled form in a dynamic link library.

(Another reason to use inheritance that may seem a bit lame is this: Suppose you're writing a book about object-oriented programming. In the previous chapter you've just presented a complete class named *Date*, and now you'd like to add a few features to it. Why make the previous class even longer than it is if you can just derive from it and show the new features that way?)

But mostly inheritance is used as an architectural mechanism that allows programmers to avoid duplicating code. Often a programmer will see that a particular job requires several classes. With further work and design, it appears that two of these classes might be able share about 75 percent of their code. It makes sense for one of these classes to inherit from the other, perhaps replacing a method or two, perhaps adding a method or two, whatever's required. Object-oriented design is an art (and a science) in itself, and this book will only scratch the surface.

With any luck, programmers who have come before you have used their skills to create object-oriented libraries that exhibit an intelligent structure of inheritance. If you'll be writing Windows applications using the Windows Presentation Foundation, you'll be working a lot with *controls*, which are visual user interface objects such as buttons, scrollbars, and so forth. These controls are declared in a vast hierarchy of inheritance, just a little bit of which is shown here:

```
Control
    ContentControl
        Label
        ButtonBase
            Button
            RepeatButton
            ToggleButton
                CheckBox
                RadioButton
    RangeBase
        ScrollBar
        ProgressBar
        Slider
```

This format is a standard way of showing inheritance. Each additional level of indentation shows another level of inheritance. For example, *ContentControl* derives from *Control*, and both *Label* and *ButtonBase* derive from *ContentControl*. (Not shown is the fact that *Control* derives from *FrameworkElement*, which derives from *UIElement*, which derives from *Visual*, which derives from *DependencyObject*, which derives from *DispatcherObject*, which derives from *Object*.)

Typically, as classes derive from other classes, they get less generalized and more specific. The Windows Presentation Foundation lets you subclass from the existing controls to make them even more specific—for example, a button that always has red italic text.

In this chapter I'll be creating a class named *ExtendedDate* that derives from the final *Date* class shown in the previous chapter.

The simplest subclassing syntax is:

```
class ExtendedDate: Date
{
}
```

The name of the *ExtendedDate* class is followed by a colon and then the name of the class that it's subclassing, in this case *Date*. *Date* is known as the base class to *ExtendedDate*. Many programmers put a space before the colon but it's not required. If you don't indicate a base class when you're declaring a class, then the class is assumed to derive from *object*. A class can inherit from only one other class. Multiple inheritance—a feature of some object-oriented languages in which a class can inherit from multiple classes—is not supported by C#.

With the simple declaration of *ExtendedDate*, a new class has been defined that contains all the non-private methods, properties, and fields in *Date*.

What *ExtendedDate* does *not* inherit from *Date* are the constructors. Constructors are not inherited. I repeat: Constructors are *not* inherited.

Because the simple *ExtendedDate* class shown above does not inherit any of the constructors in *Date*, C# automatically provides a parameterless constructor. This constructor allows you to create an *ExtendedDate* object like so:

```
ExtendedDate exdt = new ExtendedDate();
```

You can then use the *Year*, *Month*, and *Day* properties that *ExtendedDate* inherits from *Date*. But you cannot use the three-parameter constructor declared in *Date* to create an *ExtendedDate* object.

If you'd like to provide a three-parameter constructor in *ExtendedDate*, you must explicitly declare it. Once you do that, then C# no longer automatically provides a parameterless constructor, so you'll probably want to provide one was well. Here's a version of *ExtendedDate* that has the same constructors as *Date*:

```
class ExtendedDate: Date
{
    public ExtendedDate()
    {
    }
    public ExtendedDate(int year, int month, int day)
    {
        Year = year;
        Month = month;
        Day = day;
    }
}
```

Notice that the three-parameter constructor refers to the three properties defined by *Date*. That's fine because *ExtendedDate* has inherited those

properties. But a better way to declare the three-parameter constructor in *ExtendedDate* is for it to make use of the three-paremeter constructor in *Date*. This requires a special syntax:

```
class ExtendedDate: Date
{
    public ExtendedDate()
    {
    }
    public ExtendedDate(int year, int month, int day) :
                    base(year, month, day)
    {
    }
}
```

Notice the colon following the parameter list. This colon is followed by the keyword *base*, which refers to the base class of *Date*. The *base* keyword is followed by three arguments in parentheses, so it refers to the three-parameter constructor in *Date*. When the three-parameter *ExtendedDate* constructor executes, the three-parameter constructor in *Date* is called first, and then execution continues with the code in the body of the *ExtendedDate* constructor (if any).

I showed you a feature similar to this in Chapter 15 but using the keyword *this* rather than *base*. Collectively, these constructor calls are called *constructor initializers*. A constructor initializer causes another constructor in the same class or the base class to be executed before the code in the body of the constructor.

Only one constructor initializer is allowed per constructor. If you don't provide one, then one is provided for you that calls the parameterless constructor in the base class. In other words, you'll never need to provide a constructor initializer of *base()*. That initializer is generated automatically if you don't provide an alternative.

Imagine a series of classes: *Instrument* derives from *object*, and *Woodwind* derives from *Instrument* and *Oboe* derives from *Woodwind*. All these classes have parameterless constructors with no explicit constructor initializers. When you create an object of type *Oboe*, the constructor in *Oboe* first executes the parameterless constructor in *Woodwind*, which first executes the parameterless constructor in *Instrument*, which first executes the parameterless constructor in *object*, which doesn't actually do anything. The final result in this chain of nested constructor calls is that all the parameterless constructors are executed beginning with *object*, then *Instrument*, then *Woodwind*, and finally *Oboe*. This happens even if you're using a constructor in *Oboe* with parameters. If that constructor has no explicit constructor initializer, the parameterless constructors in *object*, *Instrument*, and *Woodwind* are executed before the constructor code in *Oboe*.

In summary, a constructor with no explicit constructor initializer automatically executes the parameterless constructor in the base class first. If a constructor includes a constructor initializer (either *base* or *this*) then that constructor is executed. Nothing else happens automatically.

In particular, a constructor with parameters does *not* automatically execute the parameterless constructor in the *same* class unless you specifically tell it to with a constructor initializer of *this()*.

Here's another way to think of it: Whenever your program creates an object, the parameterless constructor in *System.Object* is always executed first, followed by at least one constructor in every descendent of *object* leading up to the class that you're using to create the object.

What's interesting is that you can never *prevent* execution of some constructor in the base class. Think about it: If you want to prevent a constructor from calling the parameterless constructor in the base class, you must provide a constructor initializer. If you don't want *any* constructor in the base class to execute, you must specify a constructor initializer that uses *this* rather than *base*. But you can't do that for every constructor in your class. You'll end up with a circular chain of constructor calls, which by common sense (and the *C# Language Specification*, §10.10.1) is prohibited.

It's important for some constructor in the base class to always be executed because that's how fields are initialized. Every constructor actually begins first by setting the values of fields that have been initialized in their declarations. Then another constructor is executed, either explicitly (with a constructor initializer) or implicitly with a call to the parameterless constructor in the base class. If a constructor in the base class were never called, then the fields in the base class wouldn't be initialized, and that would probably cause problems.

The following program contains the simple definition of *ExtendedDate* that declares two constructors that call the corresponding constructor in the base class.

Inheritance.cs

```
//-----------------------------------------
// Inheritance.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class ExtendedDate : Date
{
    public ExtendedDate()
    {
    }
```

```
    public ExtendedDate(int iYear, int iMonth, int iDay) :
                                base(iYear, iMonth, iDay)
    {
    }
}

class Inheritance
{
    static void Main()
    {
        ExtendedDate dateMoonWalk = new ExtendedDate(1969, 7, 20);

        Console.WriteLine("Moon walk: {0}, Day of Year: {1}",
            dateMoonWalk, dateMoonWalk.DayOfYear());
    }
}
```

Because *ExtendedDate* derives from *Date*, the Inheritance project must include a link to the Date.cs file from the last chapter.

The *Date* class contains constructors, methods, and properties with access modifiers of *public*, and also one method (*IsConsistent*) and three fields with no access modifiers, which is the same as an access modifier of *private*. The *public* access modifier allows members of the class to be accessed from other classes. The *private* modifier prohibits access outside the class.

These restrictions also apply to derived classes. The *ExtendedDate* class can access the public *Year*, *Month*, and *Day* properties defined by *Date*, but has no access to the private *year*, *month*, or *day* fields in *Date*.

Between the two extremes of *public* and *private* is *protected*. A method, property, or field declared as *protected* is accessible in the class in which it's declared (of course) and also in any class that derives from that class. If *Date* had a method, property, or field declared as *protected*, it would normally not be accessible from outside the *Date* class, but it would be accessible to *ExtendedDate*.

Using *public*, *private*, and *protected* in an intelligent manner takes some thought and practice. Sometimes beginning programmers want to declare everything as *public* so that every class can access everything in every other class. (Microsoft Visual Basic actually fosters this attitude because everything is *public* by default.) But as a rule, classes should have a minimum of public methods. It's much easier to debug a class if other classes have a limited number of ways to affect it or access it. When you're debugging, you're sometimes like a detective solving a crime and it's helpful to say "The perpetrator could only have entered through this door or that window." Too many public members violates the concept of hiding data and prevents the class from being a black box of code.

Declaring something as *protected* means that you're giving some thought to what may be useful to inherited classes. Often the fact that something

needs to be protected is revealed only when you actually get down to coding some methods or properties in the derived class.

The file that follows contains a version of *ExtendedDate* that does more than just redefine the constructors. This class includes a new property named *CommonEraDay*. This property returns the number of days since the beginning of the Common Era. The date 1/1/1 has a *CommonEraDay* property of 1. The property also has a *set* accessor. ExtendedDate.cs is part of a project named CommonEraDayTest, which also requires a link to the Date.cs file.

ExtendedDate.cs

```
//-----------------------------------------
// ExtendedDate.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class ExtendedDate: Date
{
    // Constructors just execute constructors in base class.
    public ExtendedDate()
    {
    }
    public ExtendedDate(int iYear, int iMonth, int iDay):
                    base(iYear, iMonth, iDay)
    {
    }

    // CommonEraDay sets/gets number of days since beginning of common era.
    public int CommonEraDay
    {
        get
        {
            return DaysBeforeYear() + DayOfYear();
        }
        set
        {
            Day = 1;    // Prevent any inconsistencies during the calculation.

            // Year calculation if leap years were every four years.
            Year = (int)((value - .125m) / 365.25m) + 1;
            int iDayOfYear = value - DaysBeforeYear();

            // Adjust for leap year anomalies every hundred years.
            if (iDayOfYear > (IsLeapYear(Year) ? 366 : 365))
            {
                iDayOfYear -= IsLeapYear(Year) ? 366 : 365;
                Year++;
            }

            // Find the Month and Day
            for (Month = 12; Month > 0; Month--)
            {
```

```
                if (iDayOfYear >= DayOfYear())
                {
                    Day = iDayOfYear - DayOfYear() + 1;
                    break;
                }
            }
        }
    }

    // Private method for use by CommonEraDay property.
    int DaysBeforeYear()
    {
        int iYear = Year - 1;

        return (int) (365.25m * iYear) - iYear / 100 + iYear / 400;
    }
}
```

The code that supports the *CommonEraDay* property assumes that the leap year rules we currently observe were in effect from the beginning of the Common Era. In reality, much of the Western world celebrated leap years every four years without exception until the introduction of the Gregorian calendar in 1582. To reorient the calendar to compensate for the damage caused by the Julian calendar, 10 days were skipped. The date October 4, 1582 was followed by October 15, 1582. Algorithms that account for the switch from the Julian calendar to the Gregorian calendar are truly messy. (See Chapter 5 of Jean Meeus's *Astronomical Algorithms*, Willmann-Bell, 1991 for a taste of what's involved.) My simplified approach isn't quite as bad, but it's not perfect. Fortunately, it doesn't conk out until the year 48,702. Don't worry if you don't quite get it. The nitty-gritties of the calculation aren't really the point of this exercise.

The CommonEraDayTest project concludes with the CommonEraDay-Test.cs file, which uses *ExtendedDate* to calculate a difference between two dates.

CommonEraDayTest.cs

```
//-----------------------------------------------
// CommonEraDayTest.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class CommonEraDayTest
{
    static void Main()
    {
        Console.Write("Enter the year of your birth: ");
        int year = Int32.Parse(Console.ReadLine());

        Console.Write("And the month: ");
        int month = Int32.Parse(Console.ReadLine());
```

```
        Console.Write("And the day: ");
        int day = Int32.Parse(Console.ReadLine());

        ExtendedDate exdtBirthday = new ExtendedDate(year, month, day);
        ExtendedDate exdtMoonWalk = new ExtendedDate(1969, 7, 20);

        int daysElapsed = exdtMoonWalk.CommonEraDay -
                                exdtBirthday.CommonEraDay;

        if (daysElapsed > 0)
            Console.WriteLine(
                "You were born {0:N0} days before the moon walk.",
                daysElapsed);

        else if (daysElapsed == 0)
            Console.WriteLine(
                "You were born on the day of the moon walk.");

        else
            Console.WriteLine(
                "You were born {0:N0} days after the moon walk.",
                -daysElapsed);
    }
}
```

This program calculates the difference between two dates by subtracting their *CommonEraDay* properties. Wouldn't it be nice just to subtract one *object* from the other? That will become a reality in Chapter 20.

Chapter 19. Virtuality

As you saw in Chapter 16, concepts of equality that might have once been intuitively clear become somewhat muddied when classes are involved. Inheritance raises its own issues regarding equality and assignment. To what extent are classes that are related by inheritance equivalent to each other?

It turns out that object-oriented languages like C# provides some interesting features related to inheritance that culminate in the convenience and versatility of virtual methods.

The most basic of these features involves conversion. In particular C# provides implicit conversions from any object to any ancestral type. To explore these conversions, I'll use as an example the *ExtendedDate* class from the previous chapter. As you'll recall, this class derived from the *Date* class in Chapter 17, and the *Date* class implicitly derives from *System.Object*, also known by the C# keyword *object*. Both *object* and *Date* are ancestral types to *ExtendedDate*.

This code creates an object of type *ExtendedDate*:

```
ExtendedDate exdt = new ExtendedDate();
```

C# provides an implicit conversion from *exdt* to an object of any ancestral type. This means that you can assign *exdt* to an object declared to be of type *Date*:

```
Date dt = exdt;
```

No new object is created by this statement. The only object we're dealing with was created by the *new* expression in the preceding statement. That expression created an object of type *ExtendedDate* in the heap and returned a reference to it. Now the *dt* variable refers to the same object. This expression returns *true*:

```
exdt == dt
```

C# allows a conversion from type *ExtendedDate* to type *Date* for two reasons:

First, objects of type *ExtendedDate* and objects of type *Date* are references, and all references are the same size. The compiler has reserved space on the stack for both *exdt* and *dt*, and in both cases this space is sufficient to store a reference. There's no practical problem in copying a reference from one slot on the stack to another.

Secondly, in this particular case, C# allows the conversion because *ExtendedDate* derives from *Date*. In a very real sense, an *ExtendedDate*

object qualifies as a *Date* object because anything a *Date* object can do, an *ExtendedDate* object can also do. For example, this expression is no problem:

```
dt.DayOfYear()
```

Although *dt* really stores an *ExtendedDate* object, *ExtendedDate* has inherited the *DayOfYear* method from *Date*.

But this expression is a problem:

```
dt.CommonEraDay
```

Even though in our little example the *dt* variable is really storing an *ExtendedDate* object, and *ExtendedDate* supports the *CommonEraDay* property, the C# compiler will *not* let this expression pass. The *dt* variable is declared as an object of type *Date*, and the *Date* class does not have a *CommonEraDay* property. C# considers this expression an error.

Assigning an object to a variable of an ancestral type is sometimes known as *upcasting*, even though you don't need an explicit cast. The conversion is implicit and always allowed.

Just as you can assign an object of type *ExtendedDate* to a *Date* variable, you can declare a variable of type *object* and assign an *ExtendedDate* object to it:

```
object obj = exdt;
```

You can assign any object to this *obj* variable because every object is created from a class or structure that derives from *object*. Every object is an *object*.

The *obj* variable to which we've assigned the *ExtendedDate* object still "knows" its true nature. That information is part of what's stored in the heap along with the instance fields of the object itself. For example, you can pass *obj* to the *Console.WriteLine* method:

```
Console.WriteLine(obj);
```

You'll see the date displayed. *Console.WriteLine* effectively calls the *ToString* method of *obj*, and that's valid because *System.Object* defined the *ToString* method to begin with. *Date* redefined the *ToString* method to display the date, and *ExtendedDate* inherited that redefined *ToString* method. That the correct *ToString* method gets called is part of the pleasures associated with declaring and overriding virtual methods. That's what this chapter is all about.

Some of the following code may be a little confusing, so I want to distinguish between a variable's *declared* type and its *actual* type. Consider the *obj* variable that's been assigned an instance of the *ExtendedDate* class. Its declared type is *object* but its actual type is *ExtendedDate*. A variable's declared type never changes, and in my examples the name of the variable will indicate what that is. A variable's

actual type can change as the variable is assigned different objects. However, a variable's actual type can't be just anything. It is always the declared type or a descendant of its declared type.

C# provides an implicit conversion from any object to any ancestral type. Going the other way (*downcasting*, as it's called) requires an explicit cast:

```
ExtendedDate exdt2 = (ExtendedDate) obj;
```

Even if *obj* were never assigned an object of type *ExtendedDate*, the C# compiler would still allow this cast. Because *ExtendedDate* is derived from *object*, *obj* could conceivably be storing an object of type *ExtendedDate*, so the C# compiler awards that cast and assignment statement its good programming seal of approval.

However, there's still runtime to consider. If at runtime *obj* is not actually an *ExtendedDate* object (or an instance of a class derived from *ExtendedDate*, if such a class existed) then the assignment statement throws an *InvalidCastException*.

If the actual type of *obj* is truly *ExtendedDate*, then you can cast it and access a property in a single expression:

```
((ExtendedDate) obj).CommonEraDay
```

The double parentheses are needed because casting is a unary operation and the period is a primary operation that has higher precedence than the unary operation. Without the parentheses that extend around *obj*, the cast would seem to be applied to the value returned from the *CommonEraDay* property.

You can also cast *obj* to a *Date* object to access a property or call a method declared in *Date*:

```
((Date) obj).DayOfYear()
```

Calling *DayOfYear* would also work if you cast *obj* to *ExtendedDate*. But you wouldn't be able to cast *obj* to a class that derives from *ExtendedDate* (if such a class existed). That cast would fail at runtime, and it makes sense that it should, because a class that derives from *ExtendedDate* can do more than *ExtendedDate* and the actual type of *obj* is just *ExtendedDate*.

If *obj* isn't actually an instance of *ExtendedDate* (or a class that derives from *ExtendedDate*) and you try to cast it to an *ExtendedDate* object, you'll raise an exception. To avoid raising an exception, you can use the *as* operator instead:

```
ExtendedDate exdt2 = obj as ExtendedDate;
```

The *as* operator is similar to casting except that it doesn't raise an exception if *obj* is not an *ExtendedDate* object. In that case, the *as* operator returns *null*. (You can't use *as* to cast value types.) Programs

that use the *as* operator should be prepared for a *null* result and check for it in code:

```
if (exdt2 != null)
{
    ...
}
```

It is also possible for a program to determine the actual type of an object before the program tries to cast it. The *System.Object* class implements a method named *GetType* that is inherited by all classes and structures. *GetType* returns an object of type *Type*. I know that sounds funny, but the *System* namespace includes a class named *Type*, and that's what *GetType* returns:

```
Type typeObjVariable = obj.GetType();
```

The *Type* class has numerous properties and methods that a program can use to obtain information about the class of which *obj* is an instance, including all its properties, methods, and so forth. *GetType* will raise a *NullReferenceException* if it's applied to a *null* object.

C# also supports an operator named *typeof* that you can apply to classes, structures, enumerations, and so forth. You do not use *typeof* with objects. Like *GetType*, the *typeof* operator returns an object of type *Type*:

```
Type typeExtendedDateClass = typeof(ExtendedDate);
```

The documentation of the *Type* class in the .NET Framework says "A Type object that represents a type is unique; that is, two Type object references refer to the same object if and only if they represent the same type. This allows … for comparison of Type objects using reference equality." This means that you can use the equality operator with *Type* objects. The expression

```
obj.GetType() == typeof(ExtendedDate)
```

is *true* if the actual type of *obj* is *ExtendedDate*. If that is so, however, then the expression

```
obj.GetType() == typeof(Date)
```

returns *false*, despite the fact that *ExtendedDate* derives from *Date* and you can cast *obj* to a *Date* object.

Take a moment to nail down the difference between *GetType* and *typeof*. Both return objects of type *Type*. But *GetType* is an instance method that you call for a particular object whereas *typeof* is a C# operator that you apply to a type such as a structure or class.

You can also determine whether an object is a particular type using the *is* operator. Just as with the *as* operator, an object always appears on

the left of the *is* operator, and a type appears on the right. The expression

```
obj is ExtendedDate
```

returns *true* if *obj* is actually an instance of *ExtendedDate*. One advantage of *is* over the comparison involving *GetType* is that *is* won't raise an exception if *obj* is *null*. It will simply return *false*. Another advantage (for most applications) is that it will return *true* for any class that *ExtendedDate* derives from. If *obj* is actually an instance of *ExtendedDate*, this expression also returns *true*:

```
obj is Date
```

The *is* operator returns *true* for any type where the object on the left can be cast to the type on the right.

You can also use the *is* operator to determine whether the actual type of the object implements a particular interface. For example, the *foreach* statement works with any class that implements the *IEnumerable* interface. (To implement an interface means to include all methods that the interface declares.) If you want to determine if a particular object is an instance of a class that implements the *IEnumerable* interface, you can use the following expression:

```
obj is IEnumerable
```

The *Sytem.Object* class defines several methods that are inherited by every class and structure. In Chapter 14 you saw how a class can include a *ToString* method to provide a text representation of an object. The *ToString* method is declared as a virtual method in *System.Object*:

```
public virtual string ToString()
```

To override the *ToString* method in your own class, you use the *override* keyword:

```
public override string ToString()
```

Only methods and properties can be declared as *virtual*. Fields cannot. Any derived class can override a virtual method or property.

The *virtual* keyword is used for method and properties that are *intended* to be overridden. (Besides *ToString* you'll see in the next chapter how a class can override the virtual *Equals* and *GetHashCode* methods also declared in *System.Object*.) You can't change a method's accessibility (that is, change the method from *public* to *private*) or return type when you override a virtual method. Any virtual method overridden with *override* remains a virtual method for further descendent classes.

There will be times when you'll derive a class from an existing class and you'll want to provide a new version of a method or property that is *not* declared as virtual. Or maybe you'll want to change a method's declared access from *protected* to *public* or the other way around. Or perhaps you

need to change the return type of a method. Any member in a base class—and that includes fields as well as methods and properties—can be redefined in a derived class using the keyword *new*. This is sometimes known as *hiding* the base member.

Interestingly enough, the *new* keyword is not strictly required, but the compiler will warn you about its omission. The warning is intended to be helpful, of course. The compiler is trying to prevent you from hiding a member of the base class inadvertently.

If necessary, methods or properties in a derived class can make use of overridden methods in the base class by prefacing the method or property name with the keyword *base*. (See the *SoundEngineer* class later in this chapter for an example.) Of course, a class can reference methods in its base class that it inherits but does not override simply with the method name.

The *override* and *new* keywords have significantly different effects, as this short program demonstrates.

InheritedMethods.cs

```
//-----------------------------------------------
// InheritedMethods.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class BaseClass
{
    public virtual void VirtualMethod()
    {
        Console.WriteLine("VirtualMethod in BaseClass");
    }
    public void NonVirtualMethod()
    {
        Console.WriteLine("NonVirtualMethod in BaseClass");
    }
}

class DerivedClass : BaseClass
{
    public override void VirtualMethod()
    {
        Console.WriteLine("VirtualMethod in DerivedClass");
    }
    public new void NonVirtualMethod()
    {
        Console.WriteLine("NonVirtualMethod in DerivedClass");
    }
}
```

```
class InheritedMethods
{
    static void Main()
    {
        DerivedClass dc = new DerivedClass();
        BaseClass bc = dc;

        bc.VirtualMethod();
        bc.NonVirtualMethod();
    }
}
```

The listing begins with the definition of *BaseClass*, a class with one virtual method and one non-virtual method. Both methods simply display some text indicating what and where they are. *DerivedClass* derives from *BaseClass* and overrides the virtual method with the *override* keyword and the non-virtual method with the *new* keyword. The *Main* method creates an instance of *DerivedClass* and then assigns it to an object of type *BaseClass*:

```
DerivedClass dc = new DerivedClass();
BaseClass bc = dc;
```

The declared type of *bc* is *BaseClass*, but the actual type is *DerivedClass*. Next, *Main* calls the two methods using *bc*:

```
bc.VirtualMethod();
bc.NonVirtualMethod();
```

The results displayed by the program are interesting and reveal most of what you need to know about virtual methods:

```
VirtualMethod in DerivedClass
NonVirtualMethod in BaseClass
```

Despite the fact that the program calls these two methods with a variable declared as type *BaseClass*, the actual type of the object is *DerivedClass*. Any virtual methods you call will be those in *DerivedClass*, which is the actual type. The nonvirtual method is different. The method that's called is based on the declared type, not the actual type.

This characteristic of virtual methods is sometimes called *polymorphism*, a word derived from the Greek for "many forms." Virtual methods take on many forms as they are overridden in descendent classes.

Suppose *dt* is an instance of the *Date* class. Consider the following method call:

```
Console.WriteLine(dt);
```

If you search through the .NET Framework documentation of all the overloads of *Console.WriteLine*, you won't find one that accepts a *Date* argument. How could there be? But there *is* an overload of *WriteLine* that accepts an argument of type *object*. When you call *Console.WriteLine* with an argument of type *Date*, the C# compiler determines that the *WriteLine*

overload that comes closest is the one that has an argument of type *object*. It's acceptable because implicit casts are allowed from *Date* to *object*.

Somewhere in the body of the *WriteLine* method (or the *String.Format* method that *WriteLine* uses), the *ToString* method of the *object* parameter gets called. Because the actual type of the parameter is *Date*, and because *ToString* is a virtual method, the version of *ToString* that gets executed is the one in the *Date* class.

The difference between virtual and nonvirtual methods may become clearer when you consider the role of the compiler. Suppose your program contains a declaration of an object:

```
SomeClass someObject;
```

Or perhaps *someObject* appears in a parameter list to a method. Later on, your code contains the following method call:

```
someObject.SomeMethod();
```

But there's a little problem here. The *someObject* variable might be an instance of *SomeClass* like it's declared, or it might be an instance of a class that derives from *SomeClass*.

What's the compiler to do? (The description that follows is more conceptual than an accurate account of the compiler's actions.) The first thing it does is examine the declaration of *SomeMethod* in *SomeClass*. If *SomeMethod* is not part of *SomeClass*, the compiler examines the class that *SomeClass* inherits from, and so forth, until it finds *SomeMethod*. (If the compiler never finds *SomeMethod*, then that's a compile error.) Once the compiler finds *SomeMethod*, it checks whether the method is virtual; that is, does it have a *virtual* modifier or does it have an *override* modifier to override a virtual method in an ancestral class? If the method is not virtual, the compiler has it easy. The compiler knows exactly which method should be invoked when *SomeMethod* is called for *someObject*. It's the one that's declared right in *SomeClass* or the one that *SomeClass* inherits. The compiler can match up the code with the precise method call.

If *SomeMethod* is a virtual method, however, the compiler has a problem. The compiler doesn't have enough information to figure out which version of *SomeMethod* should be invoked. It depends on the actual type of *someObject* and, in general, that's not known at compile time. The actual type of *someObject* is known only when the program is run and *SomeMethod* is called. Only at runtime can the correct version of *SomeMethod* be invoked. Only at runtime can that call to *Tostring* in *Console.WriteLine* be hooked up to the appropriate *ToString* method. The process of hooking up a method call with a virtual method is known as *late binding* because it takes place while the program is actually running.

Virtual methods are an essential part of object-oriented programming. Without virtual methods, an expression such as

```
obj.ToString()
```

would be worthless. But because virtual methods require some additional overhead at runtime, they shouldn't be used indiscriminately.

You'll probably want to use virtual methods in situations where you have a general case, and then variations on that general case, and you want to use the same property or method names with these variations, but you want the implementations to be different.

For example, here's a general case that contains a virtual method:

```
class BaseClass
{
    public virtual int DoSomething()
    {
        ...
    }
    ...
}
```

The first variation derives from *BaseClass* and overrides the virtual method:

```
class FirstVariation: BaseClass
{
    public override int DoSomething()
    {
        ...
    }
    ...
}
```

The second variation does likewise:

```
class SecondVariation: BaseClass
{
    public override int DoSomething()
    {
        ...
    }
    ...
}
```

You can make as many of these descendent classes as you want.

Because *FirstVariation* and *SecondVariation* derive from *BaseClass*, objects of these types can be converted to objects of type *BaseClass* without casting. The implications of this simple fact are astonishing: You can store objects of type *FirstVariation* and *SecondVariation* in an array of type *BaseClass*. Or you can pass these objects to a method that has a *BaseClass* parameter. Even though you're treating these objects as if they were *BaseClass* objects, whenever you call *DoSomething*, the version

of *DoSomething* in *FirstVariation* or *SecondVariation* will execute. You can always determine what the actual type of the object is by calling the *GetType* method, but you may find it convenient to treat these objects uniformly without worrying about the actual type. And, you can later define additional descendents of *BaseClass* with minimal impact to the rest of your code.

For a more concrete example, let's look at an orchestra that pays its musicians a flat $100 per performance (public funding of the arts being what it is). Here's a simple class containing a constructor to store the musician's name, a read-only property to obtain the musician's name, and a *CalculatePay* method that returns the decimal value 100.

Musician.cs

```
//---------------------------------------
// Musician.cs (c) 2006 by Charles Petzold
//---------------------------------------
class Musician
{
    // Private field
    string strName;

    // Public property
    public string Name
    {
        get
        {
            return strName;
        }
    }

    // Constructor
    public Musician(string strName)
    {
        this.strName = strName;
    }

    // Virtual Method
    public virtual decimal CalculatePay()
    {
        return 100;
    }
}
```

Notice that *CalculatePay* is a virtual method. It's virtual because not every musician is paid $100. The harp players, for example, are paid based on the weight of their harps.

Harp.cs

```
//-------------------------------------
// Harp.cs (c) 2006 by Charles Petzold
//-------------------------------------
```

```
class Harp: Musician
{
    int weight;

    public Harp(string strName, int weight): base(strName)
    {
        this.weight = weight;
    }
    public override decimal CalculatePay()
    {
        return 1.5m * weight;
    }
}
```

The *Harp* class subclasses the *Musician* class. It declares its own constructor for both a name and a harp weight and then uses a constructor initializer to execute the constructor in *Musician* to store the harpist's name. The *Harp* class itself stores the weight of the harp. The new *CalculatePay* method has an *override* modifier and implements its own pay formula.

Violinists are paid a little more than the other musicians, but they're also penalized if they break a string during performance.

Violin.cs

```
//---------------------------------------
// Violin.cs (c) 2006 by Charles Petzold
//---------------------------------------
class Violin: Musician
{
    int numBrokenStrings;

    public Violin(string strName, int numBrokenStrings): base(strName)
    {
        this.numBrokenStrings = numBrokenStrings;
    }
    public override decimal CalculatePay()
    {
        return 125 - 50 * numBrokenStrings;
    }
}
```

The French horn is a notoriously difficult instrument, and the players are paid based on the number of correct notes and flubbed notes.

FrenchHorn.cs

```
//-----------------------------------------
// FrenchHorn.cs (c) 2006 by Charles Petzold
//-----------------------------------------
class FrenchHorn: Musician
{
    int numGoodNotes, numFlubbedNotes;
```

```
    public FrenchHorn(string strName, int numGoodNotes, int numFlubbedNotes):
        base(strName)
    {
        this.numGoodNotes = numGoodNotes;
        this.numFlubbedNotes = numFlubbedNotes;
    }
    public override decimal CalculatePay()
    {
        return 1.5m * numGoodNotes + 0.75m * numFlubbedNotes;
    }
}
```

Somehow the sound engineer has managed to get paid 125 percent of whatever the generic musician gets paid.

SoundEngineer.cs

```
//-----------------------------------------
// SoundEngineer.cs (c) 2006 by Charles Petzold
//-----------------------------------------
class SoundEngineer: Musician
{
    public SoundEngineer(string strName): base(strName)
    {
    }
    public override decimal CalculatePay()
    {
        return 1.25m * base.CalculatePay();
    }
}
```

Notice the use of the *base* keyword to reference the *CalculatePay* method in the *Musician* class.

All these classes are part of the PayTheMusicians project, which also contains the following class with the *Main* method.

PayTheMusicians.cs

```
//-----------------------------------------------
// PayTheMusicians.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;

class PayTheMusicians
{
    static void Main()
    {
        Musician[] musicians =
            {
                new Musician("Leonard"),
                new Harp("Sam", 62),
                new Violin("Sydney", 0),
                new FrenchHorn("Janet", 46, 23),
                new Musician("Chuck"),
                new Harp("Arien", 78),
                new Violin("Jason", 2),
```

```
                new FrenchHorn("Deirdre", 52, 25),
                new SoundEngineer("Fitz")
            };

        foreach (Musician mus in musicians)
            Console.WriteLine("Pay {0} the amount of {1:C}",
                mus.Name, mus.CalculatePay());
    }
}
```

The program creates nine objects based on *Musician* and its descendents, and stores all these object in an array of type *Musician*. Despite the fact that five different classes are involved here, a single array stores them all. The *foreach* statement then loops through the array, displaying the musician's name using the *Name* property and calling the *CalculatePay* method. The results reveal that each musician gets paid a correct (if not quite appropriate) amount:

```
Pay Leonard the amount of $100.00
Pay Sam the amount of $93.00
Pay Sydney the amount of $125.00
Pay Janet the amount of $86.25
Pay Chuck the amount of $100.00
Pay Arien the amount of $117.00
Pay Jason the amount of $25.00
Pay Deirdre the amount of $96.75
Pay Fitz the amount of $125.00
```

Obviously there's a lot going on behind the scenes here. Without virtual methods, tailoring calculations like this would probably require a bunch of *if* statements or a *switch*. With virtual methods, we get the same effect in a much cleaner way. You can easily add alternate pay scales by declaring new classes that derive from *Musician*. You don't have to touch the *foreach* loop or any other code that manipulates objects of type *Musician*.

Eventually, all the musicians might have special pay scales, and then you really won't be creating objects of type *Musician* at all. In that case, you can use the *abstract* modifier for the *Musician* class:

```
abstract class Musician
{
    ...
}
```

An abstract class can't be instantiated, which means the C# compiler won't allow a *new* expression involving an *abstract* class. However, you can still declare a variable of type *Musician*, and you can still have an array of type *Musician*. Everything else about the program would remain the same.

If a descendent of *Musician* doesn't override the *CalculatePay* method, the descendent ends up with the version of *CalculatePay* defined in *Musician*.

It could be that you want to declare *Musician* as an *abstract* class and also force every descendent of *Musician* to implement its own *CalculatePay* method. In that case, you can also use the *abstract* keyword for the *CalculatePay* method in *Musician*:

```
public abstract decimal CalculatePay();
```

An abstract method (or property) is implicitly virtual. Becaue this method is never called, it has no body. In such a situation, a call from the *CalculatePay* method in a derived class to the base class method (such as in the *SoundEngineer* class) would not be allowed.

A real-life example of an abstract class with abstract methods is the *Calendar* class used in conjunction with the *DateTime* structure. I'll discuss this class in Chapter 23.

The opposite of an abstract class is a sealed class. An abstract class must be subclassed to have any value to a program; a sealed class cannot be subclassed. Some classes that contain only static fields, methods, and properties, such as *Console*, *Convert*, and *Math*, are declared as static classes, which are implicitly sealed. Structures are also implicitly sealed.

And by now you should understand why inheritance isn't allowed for structures. For just one crazy moment, let's suppose you could derive one structure from another. Let's suppose you have *MyStruct1* which defines two integer fields and *MyStruct2* that derives from *MyStruct1* and defines two more integer fields. Now declare an instance of *MyStruct1*:

```
MyStruct1 ms1 = new MyStruct1();
```

As you know, *ms1* occupies 64 bytes on the stack. Now declare an instance of *MyStruct2*:

```
MyStruct2 ms2 = new MyStruct2();
```

This instance of *MyStruct2* requires 128 bytes on the stack. So what happens when you take advance of upcasting and assign *ms2* to *ms1*?

```
ms1 = ms2;
```

That's a legal assignment for classes because it copies a reference from one variable to another. But such an assignment for structures would require copying 128 bytes on the stack to an area that can only fit 64 bytes.

And that's why inheritance isn't allowed for structures.

But that's not to imply that *upcasting* isn't allowed for structures….

At the outset of this chapter, I noted that C# allows implicit conversions of objects to any ancestral type. Because every object ultimately derives from *object*, a variable declared as type *object* can be assigned any object.

An array of type *object* can store any object. A method with an *object* parameter (such as *WriteLine*) can be passed any object.

Any object. Even value types such as *int*, *decimal*, *bool* and whatever structures you declare. For example:

```
decimal pi = 3.14159m;
object obj = pi;
```

Is this right? The more you study these two simple (and completely legal) statements, the stranger they may seem. To store *pi*, the C# compiler allocates 16 bytes on the stack. For *obj*, the C# compiler allocates enough space on the stack to store a reference. Normally *obj* would be a reference to memory allocated from the heap, but the absence of a *new* expression here seems to indicate that no heap memory has been allocated.

So how can a reference like *obj* store a 16-byte *decimal*?

The answer is a little behind-the-scenes trick known as "boxing." Whenever a value type is assigned to a variable of type *object*, memory is allocated from the heap sufficient to store that value type. In this example, that's 16 bytes for the *decimal* value plus whatever overhead is required to store the object's type. The *decimal* value is then copied from the stack into the heap. That's how the *obj* variable can refer to the *decimal*.

At some point you might want to get the object back:

```
decimal m = (decimal) obj;
```

The value type is then *unboxed*. The value is extracted from the heap and copied back to the stack.

Boxing and unboxing take some time and could affect the performance of your programs. For that reason, you should be alert and wary of any code in which you are converting many value types to *object*.

Here's a little test program that reveals the performance hit of boxing. It contains two methods named *AddIntegers* and *AddObjects*, and calls each of those methods 100,000,000 times with random arguments.

TestBoxingHit.cs

```
//-----------------------------------------------
// TestBoxingHit.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Diagnostics;

class TestBoxingHit
{
    static void Main()
    {
        const int reps = 100000000;
        Random rand = new Random();
```

```
        Stopwatch watch = new Stopwatch();

        // Test method that doesn't involve boxing.
        watch.Start();
        for (int i = 0; i < reps; i++)
            AddIntegers(rand.Next(), rand.Next());
        watch.Stop();

        Console.WriteLine("Method call with no boxing: " + watch.Elapsed);

        // Test method with boxing and unboxing.
        watch.Reset();
        watch.Start();
        for (int i = 0; i < reps; i++)
            AddObjects(rand.Next(), rand.Next());
        watch.Stop();

        Console.WriteLine("Method call with boxing and unboxing: " +
                          watch.Elapsed);
    }
    static int AddIntegers(int i1, int i2)
    {
        return i1 + i2;
    }
    static int AddObjects(object obj1, object obj2)
    {
        return (int)obj1 + (int)obj2;
    }
}
```

The *AddIntegers* method requires no boxing or unboxing. The *AddObjects* method requires two boxing operations when the method is called and two unboxing operations inside the method. On my pokey machine, the *AddIntegers* calls takes a total of about 5 seconds, but the *AddObjects* calls require about 13 seconds.

If you have a method like *AddObjects* that has an *object* parameter and you're passing many value types to that method, consider writing overloads specifically for those value types. Take a look at *WriteLine*. *WriteLine* has overloads for all the C# basic types besides the version with the *object* parameter. *Writeline* tries to avoid boxing, and you should as well.

Chapter 20. Operator Overloading

The *CommonEraDay* property introduced in the *ExtendedDate* class in Chapter 18 provides a way to find the number of days between any two dates: Just subtract the *CommonEraDay* properties.

But it might be even more convenient to just subtract one *Date* object from another:

```
int numDays = dateMoonLanding – dateYourBirthday;
```

That's subtraction, but what about addition? Does it make sense to add two dates? Well, not really. But it makes a whole lot of sense to add a date and an integer. The calculation would tell you what date was a certain number of days after a particular date:

```
dateTenThousandDaysOld = dateYourBirthday + 10000;
```

This is known as operator overloading, and it's a common feature of object-oriented languages. Any class or structure that you define can specify how the standard C# operators such as plus and minus are supposed to work on objects of that type. The *String* class, for examples, defines addition as a concatenation operator. (Of course, you're not *required* to define operators for your classes; operator overloading is easily abused, so try to come up with a good justification for every operator you overload.)

The operators that a class or structure can overload are listed in the *C# Language Specification*, §7.2.2 and §10.9. The overloadable unary operators are +, –, !, ~ ++, --, *true*, and *false*. The overloadable binary operators are the arithmetic operators (+, –, *, /, and %), the logical and bitwise operators (&, |, and ^), the equality operators (== and !=), the relational operators (<, <=, >, and >=), and the shift operators (<< and >>). Because the compound assignment operators (such as +=) are defined in terms of the corresponding binary operators, you get those for free. You can't overload the conditional operators && and ||, but they will be applicable for your class if you define & and | as well as *true* and *false*. See the *C# Language Specification*, §7.11.2, for details.

At the end of this chapter I'll show you a complete class named *SuperDate* that derives from *ExtendedDate* and defines a bunch of C# operators, so the examples I'll show you will refer to *SuperDate* objects.

Operator declarations look a lot like method declarations. They always include the modifiers *public* and *static* followed by the keyword *operator* followed by the operator itself. For a binary operator like subtraction, there are two parameters, which are the object to the left of the minus

sign and the object on the right of the minus sign. At least one of the parameters must be the same type as the class. For this reason, you never need a *new* modifier when declaring an operator that's already declared in an ancestral class. The operands in the two declarations can never be the same, so they constitute distinct methods. Operators are never declared as virtual.

Here's the definition of the subtraction operator. The two parameters are objects of type *SuperDate* and the subtraction operator returns an *int*:

```
public static int operator - (SuperDate sdLeft, SuperDate sdRight)
{
    return sdLeft.CommonEraDay - sdRight.CommonEraDay;
}
```

With an operator like this defined, you'll be able to subtract one *SuperDate* object from another and get the number of days between the two dates.

When translating the *SuperDate* class to Intermediate Language, the C# compiler fabricates a static method named *op_Subtraction* that implements the subtraction operator. This is also the name that you'll see in the .NET Framework class library documentation for classes and structures that define their own operators. (Look at the documentation for the *System.Decimal* structure, for example.)

The Common Language Specification does not require that languages recognize operations such as plus and minus between non-basic types. This becomes an issue if you put the *SuperDate* class in a dynamic link library where it is accessible by any .NET language. In languages that do not allow the subtraction operator between arbitrary objects, the programmer must use the *op_Subtraction* method instead:

```
op_Subtraction(sdYourBirthday, sdMoonWalk)
```

This is not particularly attractive. Moreover, somebody might want to incorporate the *SuperDate* class in a scripting language that refers to operations with common names such as *Add* and *Subtract*.

For these reasons, it is recommended that when implementing operator overloading, you also define static methods named *Add*, *Subtract*, *Multiply*, *Divide*, and so forth that also implement these operations. Look to the *Decimal* structure for guidance.

To implement these methods, you might first define a static *Subtract* method:

```
public static int Subtract(SuperDate sdLeft, SuperDate sdRight)
{
    return sdLeft.CommonEraDay - sdRight.CommonEraDay;
}
```

You can then define the subtraction operator in terms of *Subtract*:

```
public static int operator - (SuperDate sdLeft, SuperDate sdRight)
{
    return Subtract(sdLeft, sdRight);
}
```

Or you can do it the other way around by defining *Subtract* in terms of the subtraction operator.

For the addition operator, you might first define an *Add* method that adds a *SuperDate* object and an integer to return another *SuperDate* object. Here's how the method would look:

```
public static SuperDate Add(SuperDate sdLeft, int daysRight)
{
    ...
}
```

This *Add* method needs to return an object of type *SuperDate*, so it must create an object of that type. It sounds a bit odd for a class to create an object of the class type, but there's no problem for a static method to do so. Here's a possible implementation of the *Add* method:

```
public static SuperDate Add(SuperDate sdLeft, int daysRight)
{
    SuperDate sdReturn = new SuperDate();
    sdReturn.CommonEraDay = sdLeft.CommonEraDay + daysRight;
    return sdReturn;
}
```

The method begins by creating an object of type *SuperDate* and then setting the object's *CommonEraDay* property to the calculated value of the addition. The method then returns that object.

The *Add* method would be even easier if *SuperDate* had an additional constructor—one that creates a *SuperDate* object based on an argument indicating the common era day:

```
public SuperDate(int dayCommonEra)
{
    CommonEraDay = dayCommonEra;
}
```

Now the body of the *Add* method can be written in one line:

```
public static SuperDate Add(SuperDate sdLeft, int daysRight)
{
    return new SuperDate(sdLeft.CommonEraDay + daysRight);
}
```

You can then define the plus operator in terms of the *Add* method:

```
public static SuperDate operator + (SuperDate sdLeft, int daysRight)
{
    return Add(sdLeft, daysRight);
}
```

Addition between a date and an integer is commutative, of course, but the C# compiler doesn't know that. When you want to define a commutative operation between two different types, you need two declarations of the operator to account for commutativity. Here's the second *Add* method and the second addition operator. Notice that the second *Add* method is written in terms of the first one:

```
public static SuperDate Add(int daysLeft, SuperDate sdRight)
{
    return Add(sdRight, daysLeft);
}
public static SuperDate operator + (int daysLeft, SuperDate sdRight)
{
    return Add(daysLeft, sdRight);
}
```

Now that you've implemented addition, you might want to reconsider subtraction. Besides subtracting one date from another, it makes sense to subtract an integer from a date:

```
public static SuperDate Subtract(int SuperDate sdLeft, int daysRight)
{
    return new SuperDate(sdLeft.CommonEraDay - daysRight);
}
public static SuperDate operator - (int SuperDate sdLeft, int daysRight)
{
    return Subtract(sdLeft, daysRight);
}
```

But it doesn't make sense to subtract a date from an integer.

You can also declare the two unary increment and decrement operators for *SuperDate*. There are no standard names for these operators, so you can just define the operations directly. Here's the increment:

```
public static SuperDate operator ++ (SuperDate sd)
{
    return new SuperDate(sd.CommonEraDay + 1);
}
```

The decrement is similar. There's a temptation when defining the increment and decrement to alter the parameter in the body of the method. Don't do it, or the operators won't work correctly.

You'll probably also want an equality operator. Although equality operators always return a *bool*, you still need to indicate the return type:

```
public static bool operator == (SuperDate sdLeft, superDate sdRight)
{
    return sdLeft.CommonEraDay == sdRight.CommonEraDay;
}
```

A class that includes a declaration of the equality operator must also include the inequality operator, which you can define in terms of equality:

```
public static bool operator != (SuperDate sdLeft, superDate sdRight)
{
    return !(sdLeft == sdRight);
}
```

Similarly, you can declare a relational operator:

```
public static bool operator < (SuperDate sdLeft, SuperDate sdRight)
{
    return sdLeft.CommonEraDay < sdRight.CommonEraDay;
}
```

You can then declare the opposite relational operator using logical negation:

```
public static bool operator >= (SuperDate sdLeft, SuperDate sdRight)
{
    return !(sdLeft < sdRight);
}
```

You'll want to declare the greater-than operator and the less-than-or-equal-to operator similarly.

If you declare equality and inequality operators, you'll get a warning message from the C# compiler about your failure to also declare overrides for the virtual *Equals* and *GetHashCode* methods defined in *System.Object*. As you'll recall from Chapter 16, the *Equals* method in *System.Object* implements reference equality; the *Equals* method in *System.ValueType* (from which all structures derive) implements bitwise equality, also known as value equality.

Although *SuperDate* is a class, you want the *Equals* method to return *true* when comparing two days that have the same *CommonEraDay* property. The documentation of the *Equals* method in the *System.Object* class indicates that the method must not raise an exception. In particular, if the argument to *Equals* is *null* or not of the correct type, *Equals* should simply return *false*. Here's a rather lengthy *Equals* method for *SuperDate*:

```
public override bool Equals(object obj)
{
    if (obj == null) || GetType() != obj.GetType())
        return false;

    SuperDate sd = (SuperDate) obj;
    return CommonEraDay == sd.CommonEraDay;
}
```

Here's a simpler implementation that makes use of the equality operator already declared in *SuperDate*:

```
public override bool Equals(object obj)
{
    return obj is SuperDate && this == (SuperDate) obj;
}
```

If *SuperDate* were a structure rather than a class, this method would involve a boxing and unboxing operation, and those are to be avoided. You can also define an overload of the *Equals* method whose argument is explicitly a *SuperDate* object:

```
public bool Equals(SuperDate sd)
{
    return this == sd;
}
```

System.Object also defines a static *Equals* method where the two parameters are both of type *Object*. Again, if *SuperDate* were a structure rather than a class, calling that method for two objects of type *SuperDate* would involve boxing, so you probably want to supplement that static method with one that has two explicit *SuperDate* parameters:

```
public static bool Equals(SuperDate sd1, SuperDate sd2)
{
    return sd1 == sd2;
}
```

The other virtual method in *System.Object* that the C# compiler wants you to override is *GetHashCode*, which returns a 32-bit integer. A hash code is a number that programs can use to assist in storing and retrieving objects. Two objects that are equal according to the *Equals* method must return the same integer from *GetHashCode*. However, unequal objects need not return unique hash codes. (It's certainly *preferable*, but it's not required, and it's not even possible in the general case. If the class or structure is capable of more than 2^{32} unique objects—which is the case for *long*, *double*, and *decimal*—then there are more unique objects than possible return values of *GetHashCode*.)

In general, *GetHashCode* usually performs some kind of operation on the fields of the class or structure. For example, for a *Point* structure that contains integer *x* and *y* fields, *GetHashCode* might return *x* ^ *y*. With the *SuperDate* object we've lucked out. A *SuperDate* object can be represented by a unique integer, which is the *CommonEraDay* property. *GetHashCode* can be implemented as simply as:

```
public override int GetHashCode()
{
    return CommonEraDay;
}
```

Do you want implicit or explicit conversion between *SuperDate* objects and integers? I'd shy away from implicit conversion, but explicit conversion using casting seems reasonable to me. The syntax for declaring explicit conversions involves the *explicit* keyword. Here's a method that allows an *explicit* conversion from a *SuperDate* to an *int*:

```
public static explicit operator int (SuperDate sd)
{
    return sd.CommonEraDay;
}
```

Similarly, this declaration allows explicit conversions from integers to *SuperDate* objects:

```
public static explicit operator SuperDate(int dayCommonEra)
{
    return new SuperDate(dayCommonEra);
}
```

You'd use the keyword *implicit* to declare implicit conversions.

Here's the *SuperDate* class containing all the operators I've described in this chapter.

SuperDate.cs

```
//----------------------------------------
// SuperDate.cs (c) 2006 by Charles Petzold
//----------------------------------------
using System;

partial class SuperDate: ExtendedDate
{
    // Constructors
    public SuperDate()
    {
    }
    public SuperDate(int year, int mon, int day): base(year, mon, day)
    {
    }
    public SuperDate(int dayCommonEra)
    {
        CommonEraDay = dayCommonEra;
    }

    // Equality operators
    public static bool operator == (SuperDate sdLeft, SuperDate sdRight)
    {
        return sdLeft.CommonEraDay == sdRight.CommonEraDay;
    }
    public static bool operator != (SuperDate sdLeft, SuperDate sdRight)
    {
        return !(sdLeft == sdRight);
    }

    // Relational operators
    public static bool operator < (SuperDate sdLeft, SuperDate sdRight)
    {
        return sdLeft.CommonEraDay < sdRight.CommonEraDay;
    }
```

```
    public static bool operator > (SuperDate sdLeft, SuperDate sdRight)
    {
        return sdLeft.CommonEraDay > sdRight.CommonEraDay;
    }
    public static bool operator <= (SuperDate sdLeft, SuperDate sdRight)
    {
        return !(sdLeft > sdRight);
    }
    public static bool operator >= (SuperDate sdLeft, SuperDate sdRight)
    {
        return !(sdLeft < sdRight);
    }

    // Arithmetic operators
    public static SuperDate Add(SuperDate sdLeft, int daysRight)
    {
        return new SuperDate(sdLeft.CommonEraDay + daysRight);
    }
    public static SuperDate operator +(SuperDate sdLeft, int daysRight)
    {
        return Add(sdLeft, daysRight);
    }
    public static SuperDate Add(int daysLeft, SuperDate sdRight)
    {
        return sdRight + daysLeft;
    }
    public static SuperDate operator +(int daysLeft, SuperDate sdRight)
    {
        return Add(sdRight, daysLeft);
    }
    public static int Subtract(SuperDate sdLeft, SuperDate sdRight)
    {
        return sdLeft.CommonEraDay - sdRight.CommonEraDay;
    }
    public static int operator -(SuperDate sdLeft, SuperDate sdRight)
    {
        return Subtract(sdLeft, sdRight);
    }
    public static SuperDate Subtract(SuperDate sdLeft, int daysRight)
    {
        return new SuperDate(sdLeft.CommonEraDay - daysRight);
    }
    public static SuperDate operator -(SuperDate sdLeft, int daysRight)
    {
        return Subtract(sdLeft, daysRight);
    }

    // Unary operators
    public static SuperDate operator ++ (SuperDate sd)
    {
        return new SuperDate(sd.CommonEraDay + 1);
    }
```

```
    public static SuperDate operator -- (SuperDate sd)
    {
        return new SuperDate(sd.CommonEraDay - 1);
    }

    // Explicit casts
    public static explicit operator int (SuperDate sd)
    {
        return sd.CommonEraDay;
    }
    public static explicit operator SuperDate (int daysCommonEra)
    {
        return new SuperDate(daysCommonEra);
    }

    // Overrides of methods in System.Object
    public override bool Equals(object obj)
    {
        return obj is SuperDate && this == (SuperDate) obj;
    }
    public override int GetHashCode()
    {
        return CommonEraDay;
    }
}
```

The following OperatorTest program is similar to the CommonEraTest program in the last chapter, but it uses a few of the operators declared in *SuperDate*. OperatorTest compares and subtracts the objects directly.

OperatorTest.cs

```
//-----------------------------------------
// OperatorTest.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

class OperatorTest
{
    static void Main()
    {
        Console.Write("Enter the year of your birth: ");
        int iYear = Int32.Parse(Console.ReadLine());

        Console.Write("And the month: ");
        int iMonth = Int32.Parse(Console.ReadLine());

        Console.Write("And the day: ");
        int iDay = Int32.Parse(Console.ReadLine());

        SuperDate sdBirthday = new SuperDate(iYear, iMonth, iDay);
        SuperDate sdMoonWalk = new SuperDate(1969, 7, 20);
```

```
        if (sdBirthday > sdMoonWalk)
            Console.WriteLine(
                "You were born {0:N0} days after the moon walk.",
                sdBirthday - sdMoonWalk);

        else if (sdBirthday == sdMoonWalk)
            Console.WriteLine(
                "You were born on the day of the moon walk.");

        else
            Console.WriteLine(
                "You were born {0:N0} days before the moon walk.",
                sdMoonWalk - sdBirthday);
    }
}
```

The OperatorTest project must contain SuperDate.cs as well as links to Date.cs and ExtendedDate.cs. Obviously the program doesn't perform an exhaustive test of all the operators in the *SuperDate* class, but it's a start.

You might have noticed the *partial* keyword at the very beginning of the definition of *SuperDate*. That keyword indicates that the *SuperDate* class might be more than just this one file. Parts of the *SuperDate* class might be defined in other files. For this chapter, however, *SuperDate* actually *is* just this one file. You're allowed to use the *partial* keyword when the entire class is present.

Why am I suddenly introducting the *partial* keyword when I don't need it? The problem is this:

In the past several chapters I've been using the *Date*, *ExtendedDate*, and *SuperDate* classes to demonstrate not only inheritance, but other aspects of object-oriented programming. Over the course of these chapters, I progressively added methods, constructors, properties, and operators to these classes. It was very convenient to use inheritance to break up the material into separate classes. Without inheritance, the size and contents of the overall class would have been too much to tackle in one big chapter.

In the real world, inheritance is not often used to restrict the size of a class to something short enough to be discussed in the chapter of a book. In fact, you'd probably use a structure rather than a class to represent a date. (If I were to do it over again, the only field in my *Date* structure would be a zero-based Common Era day.) Structures are more closely associated with objects that have particular numeric values; therefore operating overloading is *much* more common in structures than in classes.

Structures cannot be inherited, and that's actually an advantage when the structure contains operator declarations. Operators in classes can be problematic when the class is inherited.

For example, suppose you define another class named *HyperDate* that inherits from *SuperDate*:

```
class HyperDate: SuperDate
{
    // Nothing here yet.
}
```

In a program, you declare a *HyperDate* object like this:

```
HyperDate hdSputnik = new HyperDate(1957, 10, 4);
```

HyperDate inherits all the methods and operations declared in *SuperDate*, but some of them are no longer quite as easy to use. Try this:

```
HyperDate hd = hdSputnik + 7;    // Won't work!
```

The statement makes use of the addition operator declared in *SuperDate*. The appropriate overload has two parameters, a *SuperDate* object and an integer. Passing a *HyperDate* object to the addition operator in *SuperDate* is no problem because a *HyperDate* can be implicitly converted to a *SuperDate*. The problem is the return value, which is assigned to the *HyperDate* object. The return value of the addition operator is a *SuperDate*, and there is no implicit or explicit conversion from a *SuperDate* object to a *HyperDate*. To make such a statement work, *HyperDate* would have to declare its own addition operator.

And that's why the next chapter will not define a *HyperDate* class. But the next chapter will build on *SuperDate* and add another interesting feature.

Chapter 21. Interfaces

Suppose you have an array of *SuperDate* objects from the previous chapter and you want to sort them. You could write a sorting algorithm yourself (which would certainly be a good exercise) or you can use one of the static *Sort* methods of the *Array* class. But to use *Sort*, the elements of the array must implement the *IComparable* interface.

What does it mean to "implement the *IComparable* interface"? *IComparable* is defined in the *System* namespace, probably as simply as this:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

It starts off looking like a class or structure named *IComparable*, except that instead of *class* or *struct* the keyword *interface* appears. Then there's a method named *CompareTo* that has an *object* parameter and returns an *int*, but the method has no body.

Interfaces are entirely overhead! They contain no code. All interfaces defined in the .NET Framework begin with a capital *I*, but that's just a convention. You can name your own interfaces anything you like.

An interface is generally a collection of methods without bodies, although interfaces can also contain other types of members such as properties (which must also have empty bodies). The *IComparable* interface has a single method named *CompareTo* that returns an *int*. The documentation of *CompareTo* provides a set of rules:

- If the instance is less than the parameter, *CompareTo* returns a negative number.
- If the instance is equal to the parameter, *CompareTo* returns 0.
- If the instance is greater than the parameter, *CompareTo* returns a positive number.
- If the parameter is *null*, *CompareTo* returns a positive number.
- If the parameter is the wrong type, *CompareTo* throws an *ArgumentException*.

You'll notice that all the basic numeric types and the *String* class implement the *IComparable* interface.

To make *SuperDate* implement the *IComparable* interface, you start at the very top of the *SuperDate* class declaration and list *IComparable*

along with *ExtendedDate*, which is the class that *SuperDate* inherits from:

```
class SuperDate: ExtendedDate, IComparable
```

A class can derive from only one other class, but a class (or structure) can implement multiple interfaces, which must be separated by commas.

A class that implements an interface must include all the methods in that interface. To implement the *IComparable* interface, *SuperDate* must declare a *CompareTo* method in accordance with the rules listed earlier.

Here's a file named SuperDate2.cs that contains a partial class definition of *SuperDate* that implements the *IComparable* interface by including the *CompareTo* method.

SuperDate2.cs

```
//-----------------------------------------
// SuperDate2.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;

partial class SuperDate : ExtendedDate, IComparable
{
    public int CompareTo(object obj)
    {
        if (obj == null)
            return 1;

        if (!(obj is SuperDate))
            throw new ArgumentException();

        return this - (SuperDate)obj;
    }
}
```

This file is intended to supplement the partial *SuperDate* class definition in the SuperDate.cs file from the last chapter.

One reason for implementing *IComparable* is to easily sort arrays using the static *Array.Sort* method. Because arrays can contain any type of object, *Sort* needs some way to determine if one element of the array is less than, equal to, or greater than another element. *Sort* can't figure that out on its own. For that reason, *Sort* requires that its argument be an array whose elements implement the *IComparable* interface. What the *Sort* method *really* wants is to call *CompareTo* on the elements of the array. That's the only way *Sort* knows how to sort an array of objects it's not familiar with.

Let's try it out. The following program contains an array of famous composers alphabetized by last name. A corresponding array of *SuperDate* objects contains the birth dates of these composers. The program sorts the arrays by birth date and displays the results.

The DateSorting project includes SuperDate2.cs and DateSorting.cs and has links to Date.cs, ExtendedDate.cs, and SuperDate.cs.

DateSorting.cs

```
//-----------------------------------------
// DateSorting.cs (c) 2006 by Charles Pezold
//-----------------------------------------
using System;

class DateSorting
{
    static void Main()
    {
        string[] strComposers =
            {
                "John Adams",         "Johann Sebastian Bach",
                "Bela Bartok",        "Ludwig van Beethoven",
                "Hector Berlioz",     "Pierre Boulez",
                "Johannes Brahms",    "Benjamin Britten",
                "Aaron Copland",      "Claude Debussy",
                "Philip Glass",       "George Frideric Handel",
                "Franz Joseph Haydn", "Gustav Mahler",
                "Claudio Monteverdi", "Wolfgang Amadeus Mozart",
                "Sergei Prokofiev",   "Steve Reich",
                "Franz Schubert",     "Igor Stravinsky",
                "Richard Wagner",     "Anton Webern"
            };
        SuperDate[] sdBirthDates =
            {
                new SuperDate(1947, 2, 15),  new SuperDate(1685, 3, 21),
                new SuperDate(1881, 3, 25),  new SuperDate(1770, 12, 17),
                new SuperDate(1803, 12, 11), new SuperDate(1925, 3, 26),
                new SuperDate(1833, 5, 7),   new SuperDate(1913, 11, 22),
                new SuperDate(1900, 11, 14), new SuperDate(1862, 8, 22),
                new SuperDate(1937, 1, 31),  new SuperDate(1685, 2, 23),
                new SuperDate(1732, 3, 31),  new SuperDate(1860, 7, 7),
                new SuperDate(1567, 5, 15),  new SuperDate(1756, 1, 27),
                new SuperDate(1891, 4, 23),  new SuperDate(1936, 10, 3),
                new SuperDate(1797, 1, 31),  new SuperDate(1882, 6, 17),
                new SuperDate(1813, 5, 22),  new SuperDate(1883, 12, 3)
            };

        Array.Sort(sdBirthDates, strComposers);

        for (int i = 0; i < strComposers.Length; i++)
            Console.WriteLine("{0} was born on {1}.",
                strComposers[i], sdBirthDates[i]);
    }
}
```

The *Array* class contains many overloads of the static *Sort* method. The simplest version simply sorts a single array, which is not quite good enough for this program. We need the second simplest sort, which has *two* array arguments. The *Sort* method rearranges both arrays in the

same way based on a sort of the elements in the first array. When two arrays are used in such a way, the elements of the first array are sometimes referred to as *keys*. Here's the result:

```
Claudio Monteverdi was born on 15 May 1567.
George Frideric Handel was born on 23 Feb 1685.
Johann Sebastian Bach was born on 21 Mar 1685.
Franz Joseph Haydn was born on 31 Mar 1732.
Wolfgang Amadeus Mozart was born on 27 Jan 1756.
Ludwig van Beethoven was born on 17 Dec 1770.
Franz Schubert was born on 31 Jan 1797.
Hector Berlioz was born on 11 Dec 1803.
Richard Wagner was born on 22 May 1813.
Johannes Brahms was born on 7 May 1833.
Gustav Mahler was born on 7 Jul 1860.
Claude Debussy was born on 22 Aug 1862.
Bela Bartok was born on 25 Mar 1881.
Igor Stravinsky was born on 17 Jun 1882.
Anton Webern was born on 3 Dec 1883.
Sergei Prokofiev was born on 23 Apr 1891.
Aaron Copland was born on 14 Nov 1900.
Benjamin Britten was born on 22 Nov 1913.
Pierre Boulez was born on 26 Mar 1925.
Steve Reich was born on 3 Oct 1936.
Philip Glass was born on 31 Jan 1937.
John Adams was born on 15 Feb 1947.
```

Try swapping the order of the arguments to the *Array.Sort* method like this:

```
Array.Sort(strComposers, sdBirthDates);
```

Recompile and run the program. The *String* class also implements the *IComparable* interface, and now the composers are sorted by name, albeit by the first name rather than the last name.

To sort by last name, you'll probably want a class or structure (named *Name*, for example), which has two *String* properties named *FirstName* and *LastName*, and which also implements the *IComparable* interface. The *CompareTo* method in *Name* would make use of the *CompareTo* method of the *LastName* property.

If you're using .NET 2.0 or later, you'll also notice that besides the IComparable interface, the documentation also lists something called the "IComparable Generic Interface." I'll discuss generics in Chapter 27.

Chapter 22. Interoperability

There may come a time when you're writing a .NET class and you need something that's provided by the Windows application programming interface (API) but which isn't available in any .NET class. Or, maybe you have a bunch of dynamic link libraries (DLLs) that weren't written in .NET but which you'd like to use in your .NET programs.

The example I'm going to show you in this chapter does *not* fit into any of those categories, but it will demonstrate anyway using *platform invoke* (sometimes appreviated *PInvoke*), which lets you get at Win32 API functions from your .NET programs.

Although you can use interoperability from any C# program, generally you'll want to tuck the code away in a class and provide a "wrapper" for it. In this example, I'm going to extend *SuperDate* once again and provide a static method named *Today* that returns a *SuperDate* object for today's date. The .NET *DateTime* structure provides a static property named *Now* that provides this information but I'm going to ignore that and instead call the Win32 *GetSystemTime* function, passing to it a SYSTEMTIME structure.

Many of the classes you use for interoperability are defined in the *System.Runtime.InteropServices* namespace, so you'll probably want a *using* directive for that namespace.

The *GetSystemTime* function is defined in C syntax in the Win32 documentation like this:

```
void GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

The single argument is a pointer to a SYSTEMTIME structure. The SYSTEMTIME structure is defined in C syntax like this:

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

The WORD type definition is a 16-bit unsigned integer. The first step is to define a class or structure that resembles this C structure, but using C# syntax. Here's a possibility:

```
[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
    public ushort wSecond;
    public ushort wMilliseconds;
}
```

I've kept the same field names but made them *ushort* and *public*. I made this a class rather than a structure with a name of *SystemTime*. Before the class definition is some information in square brackets. This is known as an *attribute*. Attributes are information you can attach to a type or member of a type. The information is stored as metadata along with the compiled code.

StructLayoutAttribute is a class defined in the *System.Runtime.InteropServices* namespace and is used to describe how the fields of the class or structure should be interpreted. You could supply *LayoutKind.Explicit* and then give byte offsets for all the fields.

Your code also needs a declaration of the Win32 function you need to call, in this case *GetSystemTime*:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime st);
```

Notice the *extern* keyword, which means that it's external to the program. The attribute indicates the dynamic link library in which the function is stored.

Here's a partial *SuperDate* class that contains a static method named *Today* that calls *GetSystemTime*.

SuperDate3.cs

```
//-----------------------------------------
// SuperDate3.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;
using System.Runtime.InteropServices;

partial class SuperDate
{
    [StructLayout(LayoutKind.Sequential)]
    class SystemTime
    {
        public ushort wYear;
        public ushort wMonth;
        public ushort wDayOfWeek;
        public ushort wDay;
```

```
        public ushort wHour;
        public ushort wMinute;
        public ushort wSecond;
        public ushort wMilliseconds;
    }

    [DllImport("kernel32.dll")]
    static extern void GetSystemTime(SystemTime st);

    public static SuperDate Today()
    {
        SystemTime systime = new SystemTime();
        GetSystemTime(systime);
        return new SuperDate(systime.wYear, systime.wMonth, systime.wDay);
    }
}
```

The *Today* method creates a new object of type *SystemTime*, calls the Win32 *GetSystemTime* function, and then creates a *SuperDate* object from fields of the *SystemTime* class.

Here's a program that uses this static method to display today's date.

GetTodaysDate.cs

```
//-------------------------------------------
// GetTodaysDate.cs (c) 2006 by Charles Petzold
//-------------------------------------------
using System;

class GetTodaysDate
{
    static void Main()
    {
        SuperDate sdToday = SuperDate.Today();
        Console.WriteLine("Today's date is " + sdToday);
    }
}
```

The GetTodaysDate project also includes the previous *SuperDate* files, ExtendedDate.cs, and Date.cs.

In SuperDate3.cs, you can alternatively define *SystemTime* as a structure rather than a class:

```
struct SystemTime
```

In that case, you want to pass a reference to the structure to the *GetSystemTime* function:

```
static extern void GetSystemTime(out SystemTime st);
```

Notice the use of the *out* keyword, which means that a reference is passed to the function, but the object doesn't have to be initialized first. The first two statements of *Today* could then be:

```
SystemTime systime;
GetSystemTime(out systime);
```

If you need help defining signatures for Win32 structures and function calls for use with *PInvoke*, you'll be pleased to know that there's a Wiki that contains much useful information:

http://www.pinvoke.net

Now that you've seen how you can implement your own class to store dates, let's examine how the designers of the .NET Framework did it.

Chapter 23. Dates and Times

Throughout the .NET Framework, a particular moment in time is represented by an object of type *DateTime*, a structure defined in the *System* namespace. You generally create an object of type *DateTime* in one of two ways. You can use one of the constructors of *DateTime* to create an object representing a particular date and time, or you can use one of three properties (*Now*, *UtcNow*, and *Today*) to create a *DateTime* object representing the current time or date.

Here's one of the *DateTime* constructors:

```
DateTime dt = new DateTime(2007, 8, 29, 15, 30, 0);
```

The six arguments of the constructor are the year, month, date, hour, minute, and second. This constructor specifies a time and date of 3:30 PM on August 29, 2007.

DateTime constructors are also defined to specify just the year, month, and day, or to include milliseconds along with the time.

The year argument can range from 1 through 9999, the month can range from 1 through 12, the day can range from 1 to 31, the hour can range from 0 through 23, the minute and second arguments can range from 0 through 59, and the milliseconds argument can range from 0 through 999. Anything outside these ranges raises an *ArgumentOutOfRange-Exception*.

The *DateTime* structure defines two static read-only fields of type *DateTime* named *MinValue* (representing midnight on January 1 in the year 1) and *MaxValue* (one millisecond prior to midnight on January 1 in the year 10,000). If you create a *DateTime* object with a parameterless constructor, it will represent a date and time equal to *DateTime.MinValue*.

The *DateTime* constructor also throws an exception if the combination of year, month, and day arguments isn't consistent. A month argument of 2 and a day of 29 is acceptable only for a leap year. The *DateTime* structure uses leap year rules associated with the Gregorian calendar, which was instituted by Pope Gregory XIII in 1582 and eventually adopted worldwide in the years and centuries that followed. In the Gregorian calendar, a year is a leap year if it is divisible by 4 but not divisible by 100 unless it is divisible by 400. The year 1900 was not a leap year, but the year 2000 was.

The calendar in effect prior to the adaptation of the Gregorian calendar is now known as the Julian calendar, which was introduced during the

reign of Julius Caesar. Leap years were celebrated every four years without exception. The *DateTime* constructor observes Gregorian leap year rules even for years preceding the invention of the Gregorian calendar. There is no way to use *DateTime* to specify years prior to the Common Era.

DateTime defines nine get-only properties of type *Int32* that indicate the date and time represented by the *DateTime* object. These properties are named *Year*, *Month*, *Day*, *Hour*, *Minute*, *Second*, *Millisecond*, *DayOfWeek* (with values ranging from 0 for Sunday through 6 for Saturday), and *DayOfYear* (which values ranging from 1 through 366).

Let me emphasize that these properties are get-only. Once created, a *DateTime* object is immutable.

DateTime includes three static properties that let you create an object of type *DateTime* representing the current date and time. The *Now* property creates a *DateTime* object with the current local date and time:

```
DateTime dtLocal = DateTime.Now;
```

The static *Today* property creates a *DateTime* object with the current local date but with the *Hour*, *Minute*, *Second*, and *Millisecond* properties all set to zero:

```
DateTime dateToday = DateTime.Today;
```

This code is equivalent to:

```
DateTime dateToday = DateTime.Now.Date;
```

The *Date* instance property creates a new *DateTime* object based on an existing *DateTime* object but with the *Hour*, *Minute*, *Second*, and *Millisecond* properties set to zero.

The static *UtcNow* property returns a *DateTime* property for the current date and time in Coordinated Universal Time (UTC):

```
DateTime dtUtc = DateTime.UtcNow;
```

As you know, governments around the world have defined numerous time zones so that local time is calculated as hour or half-hour offsets from UTC, which is basically what was once called Greenwich Mean Time (GMT).

Greenwich, England, has played an important role in the evolution of time standards because it is the site of the Royal Greenwich Observatory (RGO), which was founded in 1675 to develop techniques of astronomical navigation for ships at sea. In the 1760s, the observatory began publishing nautical almanacs that for convenience placed the prime meridian (the line of 0° longitude) at Greenwich. This system of meridians was eventually agreed upon as a world standard in 1884, although the French continued to use Paris as the prime meridian until 1911.

Greenwich Mean Time began in 1833 when the observatory started dropping a ball that was visible to ships in the Thames every day at 1:00 p.m. In the 1840s, GMT was declared the standard time for all of Great Britain to replace various time zones that had developed over the years.

These days, the use of the term Coordinated Universal Time is considered more scientifically correct than Greenwich Mean Time. Coordinated Universal Time is abbreviated UTC as something of a compromise between the English word order—which would imply the abbreviation CUT—and the French *Temps Universel Coordonné*, which would have the abbreviation TUC. By international agreement since 1972, UTC is the same all of the world.

The length of a UTC second is based on an atomic standard rather than astronomical observations. Because the rotation of the earth has been gradually slowing down, occasionally it is necessary to adjust UTC to keep it in sync with astronomical solar time. This is done with the introduction of leap seconds.

The system date and time that Windows maintains is UTC rather than local time, and these days that time is usually set (and automatically corrected) from an Internet time server. Windows also maintains a current time zone, which the user sets from the Date/Time properties dialog opened from the Control Panel or by double-clicking the time on the Windows taskbar. Whenever a Windows application requires a local time, that time is calculated from UTC based on the current time zone. Maintaining the system time in UTC rather than local time (as was once the case) makes it easier for Windows to adjust for changes in time zone or daylight saving time.

Daylight saving time is that quaint custom observed in many locales of changing the clocks twice a year. The principle behind it is simple: As the summer solstice approaches, the sun is rising earlier and setting later, so it's no big deal to get out of bed a little earlier and enjoy even more sun in the evening. Some countries observe daylight saving time and some don't and those that observe it frequently begin and end it on different dates. Even within some countries, notably the United State, daylight saving time is implemented inconsistently, sometimes even within the same state. The Date/Time properties dialog lets a user specify whether Windows is to automatically adjust for daylight saving time.

In .NET 1.X, there was nothing in the *DateTime* structure that implied whether a particular *DateTime* object represented UTC or local time. A program using this structure had to keep track of that particular information itself.

In .NET 2.0, a new get-only property was introduced in the *DateTime* structure named *Kind*, of type *DateTimeKind*, an enumeration that contains the three members *Unspecified*, *Local*, and *Utc*.

If you create a *DateTime* object using the static *Now* or *Today* properties, the *Kind* property will equal *DateTimeKind.Local*. In contrast, the static *DateTime.UtcNow* property returns a *DateTime* object with the *Kind* property set to *DateTimeKind.Utc*.

If you create a *DateTime* object using the constructor shown above:

```
DateTime dt = new DateTime(2007, 8, 29, 15, 30, 0);
```

then the *Kind* property is set to *DateTimeKind.Unspecified*. However, .NET 2.0 also introduced additional constructors that let you specify the kind of time:

```
DateTime dtLocal = new DateTime(2007, 8, 29, 15, 30, 0,
                                DateTimeKind.Local);
```

This *DateTime* object will have a *Kind* property of *DateTimeKind.Local*.

DateTime defines two instance methods named *ToLocalTime* and *ToUniversalTime* that convert between UTC and local time. For example:

```
DateTime dtUtc = dtLocal.ToUniversalTime();
```

The *ToLocalTime* and *ToUniversalTime* methods examine the *Kind* property of the *DateTime* object and do not perform a conversion if the time is already of the desired kind. For example, consider the following code:

```
DateTime dtNew = dt.ToLocalTime();
```

If the *Kind* property of dt equals *DateTimeKind.Unspecified* or *DateTime-Kind.Utc*, then the *ToLocalTime* method returns a local time under the assumption that *dt* represents a *Utc* time. However, if the *Kind* property of *dt* is *DateTimeKind.Local*, then *ToLocalTime* returns the same time as encoded in *dt*.

Like all the properties of *DateTime*, the *Kind* property is get-only. If you need to change the *Kind* property of a particular *DateTime* object, you can use the static *SpecifyKind* method to create a new *DateTime* object:

```
DateTime dtLocal = DateTime.SpecifyKind(dt, DateTimeKind.Local);
```

Tick Counts

Another way of representing date and time is by a number of 100-nanosecond clock ticks. Internally, the *DateTime* structure stores the date and time as the number of ticks since midnight, January 1, of the Common Era year 1. This value can be obtained from the get-only *Ticks* property of type *long*.

For a *DateTime* object representing midnight on the date January 1, 2001, the *Ticks* property returns the value 631,139,040,000,000,000. There are 10,000 ticks in a millisecond, 10,000,000 ticks in a second, 600,000,000 ticks in a minute, 36,000,000,000 ticks in an hour, and

864,000,000,000 ticks in a day. That means that 730,485 days have elapsed in those 2000 years, for an average of 365.2425 days per year.

The value of 365.2425 days per year is correct for the Gregorian calendar. Most years have 365 days. An extra day every four year add 0.25 to the average days per year. Excluding an extra day every 100 years lessens the average days per year by 0.01. Including an extra day every 400 years increases the average days per year by 0.0025. In other words,

```
365 + ¼ - 1/100 + 1/400 = 365.2425
```

The *DateTime* structure defines a constructor that lets you create a *DateTime* object from the number of ticks since the date 1/1/1. An additional constructor includes a *DateTimeKind* argument.

DateTime Calculations and *TimeSpan*

The *DateTime* structure contains a number of methods and overloaded operators that let you perform calculations on dates and times. The comparison operators (==, !=, <, >, <=, and >=) are all valid for *DateTime* objects. Addition and subtraction are also supported.

If you subtract one *DateTime* object from another, the result is an object of type *TimeSpan*:

```
TimeSpan ts = dt1 – dt2;
```

TimeSpan is another structure defined in the *System* namespace that represents an elapsed time in units of 100-nanoseconds, which are the same units as the *Ticks* property of *DateTime*. *TimeSpan* also has a *Ticks* property, and a constructor that accepts an argument in units of 100-nanoseconds.

The *Ticks* properties in *DateTime* and *TimeSpan* may seem similar, but it's important to keep them distinct. The *Ticks* property of *DateTime* is always the number of 100-nanosecond intervals since January 1, 1 C.E. A *TimeSpan* object represents a period of elapsed time, so the *Ticks* property of is the number of 100-nanoseconds between two points in time. The following expression, which involves an implicit *TimeSpan* object on the right side of the equality operator, always returns *true*:

```
dt.Ticks == (dt – DateTime.MinValue).Ticks
```

The *Ticks* property of a *TimeSpan* object can be negative; the *Ticks* property of *DateTime* is always non-negative.

TimeSpan defines several constructors that let you specify a certain number of days, hours, minutes, seconds, and milliseconds. For example, the following constructor returns a *TimeSpan* object representing a duration of 40 days, 30 hours, 20 minutes, and 10 seconds:

```
TimeSpan ts = new TimeSpan(40, 30, 20, 10);
```

There are no TimeSpan constructors involving months or years because months and years don't have a fixed number of days. The values in the *TimeSpan* constructors indicate a certain number of days, a number of hours, and so forth. They aren't restricted like the values in the *DateTime* constructors, and they can be negative. The following statement is perfectly legal:

```
TimeSpan ts = new TimeSpan(4000, -3000, -2000, 1000);
```

TimeSpan defines ten get-only properties besides *Ticks*. Regardless of the arguments passed to *TimeSpan* object, the *Days* property is a whole number of days. The *Hours* property ranges from 0 through 23; the *Minutes* and *Seconds* properties range from 0 through 59, and the *Milliseconds* property ranges from 0 through 999.

The remaining five *TimeSpan* properties are all of type *double* and provide the *TimeSpan* object in whatever units you want. The properties are named *TotalDays*, *TotalHours*, *TotalMinutes*, *TotalSeconds*, and *TotalMilliseconds*. The values are calculated by dividing the *Ticks* property by constant fields named *TicksPerDay*, *TicksPerHour*, and so forth.

Calendars Around the World

Four *DateTime* constructors have arguments of type *Calendar*, for example:

```
new DateTime(year, month, day, cal);
```

The final argument is of type *Calendar* and indicates how the *year*, *month*, and *day* arguments are to be interpreted. Constructors without the *Calendar* argument are assumed to refer to dates in the Gregorian calendar. These other constructors allow the arguments to refer to dates in other calendars.

Calendar is an abstract class defined in the *System.Globalization* namespace, which also includes classes that derive from *Calendar*:

Object
- *Calendar*
  - *EastAsianLunisolarCalendar*
  - *GregorianCalendar*
  - *HebrewCalendar*
  - *HijriCalendar*
  - *JapaneseCalendar*
  - *JulianCalendar*
  - *KoreanCalendar*
  - *PersianCalendar*
  - *TaiwanCalendar*

ThaiBuddhistCalendar
UmAlQuraCalendar

When you include a *Calendar* object as the last argument to the *DateTime* constructor, different consistency rules are applicable. For example,

```
new DateTime(1900, 2, 29)
```

generates an exception because 1900 isn't a leap year in the Gregorian calendar. However,

```
new DateTime(1900, 2, 29, new JulianCalendar())
```

doesn't cause an exception because in the Julian calendar every year divisible by 4 is a leap year.

Moveover, if you actually create that *DateTime* object using the *JulianCalendar* object and then look at the individual properties of the DateTime structure, you'll find that *Month* equals 3 (March) and Day equals 13. The *Year*, *Month*, and *Day* properties of the *DateTime* structure always represent dates in the Gregorian calendar. The constructor effectively converts a date in a particular calendar into a tick count; the *DateTime* properties convert from that tick count to dates in the Gregorian calendar.

The original adoption of the Gregorian calendar caused the date after October 4, 1582 to be October 15, 1582, effectively skipping 10 days. If you call

```
new DateTime(1582, 10, 5, new JulianCalendar())
```

the resultant *Month* property of the *DateTime* object will be 10 and the *Day* property will indeed by 15.

It gets more interesting. Suppose you call

```
new DateTime(5762, 5, 20, new HebrewCalendar())
```

That's the 20th day in the month of Shevat in the year 5762 of the Hebrew calendar. The resultant *DateTime* structure has a *Year* property of 2002, and *Month* and *Day* properties both equal to 2. Basically what you have here is a conversion from the Hebrew calendar to the Gregorian calendar. When the last argument to the *DateTime* constructor is a *HebrewCalendar* object, the *Month* argument can be set to 13 in some years.

Similarly, you can specify a date in the Islamic calendar:

```
new DateTime(1422, 11, 20, new HijriCalendar())
```

That's the 20th day of the month of Dhu'l-Qa'dah in the year 1422. Again, the resultant *DateTime* structure has a *Year* property of 2002 and *Month* and *Day* properties both equal to 2.

To convert from a Gregorian date to another calendar, you need to create an instance of the particular calendar, for example,

```
HebrewCalendar hebrewcal = new HebrewCalendar();
HijriCalendar hijrical = new HijriCalendar();
```

You'll also need a *DateTime* object:

```
DateTime dt = new DateTime(2002, 2, 2);
```

To convert this Gregorian date into a date in the Hebrew or Islamic calendar, call the *GetYear*, *GetMonth*, and *GetDayOfMonth* methods defined by *Calendar* and inherited by *HebrewCalendar* and *HijriCalendar*, passing to them the *DateTime* object to be converted. For example, the expression

```
hijrical.GetYear(dt)
```

returns 1422.

A Readable Rendition

Some of the most important methods in *DateTime* are those that format the date and time into human-readable form. The *DateTime* formatting includes the user's preferred cultural settings, including separators and month names and day-of-the-week names in the user's language.

When displaying dates and times, you generally want formatting to be culturally specific. However, sometimes that's undesirable. Sometimes dates and times must be embedded in documents that must be viewed by people in multiple cultures or merged with similar documents. In this case, a program should use a consistent date and time format, perhaps in accordance with some international standard. In the jargon of the .NET Framework, such formats are said to be *culture-invariant*.

The *ToString* method define by *DateTime* has a no-argument version, of course, but also overloads that accept a formatting string, or an instance of a class that implements the *IFormatProvider* interface, or both. For formatting *DateTime* objects, the appropriate class that implements *IFormatProvider* is *DateTimeFormatInfo*, which is in the *System.Globalization* namespace. *DateTimeFormatInfo* has two static properties named *CurrentInfo* and *InvariantInfo* that returns instances of *DateTimeFormatInfo*.

The following program show combinations of formatting strings and *DateTimeFormatInfo* objects to format the current date and time.

DateAndTimeFormatting.cs

```
//-------------------------------------------------------
// DateAndTimeFormatting.cs (c) 2006 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Globalization;
```

```
class DateAndTimeFormatting
{
    static DateTime dt = DateTime.Now;

    static void Main()
    {
        ShowFormatting(DateTimeFormatInfo.InvariantInfo, "InvariantInfo");
        ShowFormatting(DateTimeFormatInfo.CurrentInfo, "CurrentInfo");
    }
    static void ShowFormatting(DateTimeFormatInfo format, string strLabel)
    {
        Console.WriteLine(strLabel);
        Console.WriteLine(new string('-', strLabel.Length));

        string[] strFormats = {"d", "D", "f", "F", "g", "G", "m",
                               "r", "s", "t", "T", "u", "U", "y" };

        foreach (string strFormat in strFormats)
            Console.WriteLine("{0}: {1}", strFormat,
                              dt.ToString(strFormat, format));
        Console.WriteLine();
    }
}
```

Notice the *strFormats* array in the *ShowFormatting* method. That array contains the formatting strings you can use in the *ToString* method. (You can use those same letters in the placeholders in the formatting string of *Console.WriteLine*.) The program first shows the formatting for *DateTimeFormatInfo.InvariantInfo*:

```
InvariantInfo
-------------
d: 12/02/2006
D: Saturday, 02 December 2006
f: Saturday, 02 December 2006 16:48
F: Saturday, 02 December 2006 16:48:43
g: 12/02/2006 16:48
G: 12/02/2006 16:48:43
m: December 02
r: Sat, 02 Dec 2006 16:48:43 GMT
s: 2006-12-02T16:48:43
t: 16:48
T: 16:48:43
u: 2006-12-02 16:48:43Z
U: Saturday, 02 December 2006 21:48:43
y: 2006 December
```

I have my regional settings set for American English, so the following formatting is shown for *DateTimeFormatInfo.CurrentInfo*:

```
CurrentInfo
-----------
d: 12/2/2006
D: Saturday, December 02, 2006
```

```
f: Saturday, December 02, 2006 4:48 PM
F: Saturday, December 02, 2006 4:48:43 PM
g: 12/2/2006 4:48 PM
G: 12/2/2006 4:48:43 PM
m: December 02
r: Sat, 02 Dec 2006 16:48:43 GMT
s: 2006-12-02T16:48:43
t: 4:48 PM
T: 4:48:43 PM
u: 2006-12-02 16:48:43Z
U: Saturday, December 02, 2006 9:48:43 PM
y: December, 2006
```

The letters are mnemonics of sorts:

Letter	Mnemonic
d	Date
f	Full
g	General
m	month/day
r	RFC
s	sortable
t	time
u	universal
y	year/month

When the uppercase and lowercase letters produce different result (such as *d* and *D*)the uppercase letter produces a longer string. For the *r*, *R*, *s*, or *u* formatting strings, the results are the same regardless of the second argument to *ToString*. (You can also define your own formatting.)

The *ToString* method with a *null* or an absent formatting string argument is the same as *G*. *ToString* without a *DateTimeFormatInfo* argument is the same as *DateTimeFormatInfo.CurrentInfo*.

Using *r* or *R* results in the RFC 1123 format. (RFC stands for Request for Comments. RFCs are documentations of Internet standards and are obtainable from many sources, including the Web site of the Internet Engineering Task Force, http://www.ietf.org.) The *s* format is known as ISO 8601 format, and is intended to be universal and easily sortable. The *T* in the center is known as a *time designator* and separates the date and time. Dates that begin with months or days of the month can't be sorted quite as easily. The *u* format is similar to *s* except that the time designator is missing and the string ends with a *Z*. In military and radio circles, UTC is sometimes known as *Zulu time*, Zulu being used to represent *Z*, and *Z* referring to zero degrees of longitude.

The *U* format option performs a conversion to UTC if the *DateTime* value is a local time.

The *DateTime* structure has four other convenient culture-specific formatting methods:

- *ToShortDateString* is equivalent to *d* formatting.
- *ToLongDateString* is equivalent to *D* formatting.
- *ToShortTimeString* is equivalent to *t* formatting.
- *ToLongTimeString* is equivalent to *T* formatting.

Now go into your Regional Options dialog and change the locale to France. The *InvariantInfo* formatting is the same, but the *CurrentInfo* results are demonstrably different:

```
CurrentInfo
-----------
d: 02/12/2006
D: samedi 2 décembre 2006
f: samedi 2 décembre 2006 17:07
F: samedi 2 décembre 2006 17:07:24
g: 02/12/2006 17:07
G: 02/12/2006 17:07:24
m: 2 décembre
r: Sat, 02 Dec 2006 17:07:24 GMT
s: 2006-12-02T17:07:24
t: 17:07
T: 17:07:24
u: 2006-12-02 17:07:24Z
U: samedi 2 décembre 2006 22:07:24
y: décembre 2006
```

Chapter 24. Events and Delegates

Suppose you have a scenario with two classes with the names of *A* and *B*. Class *A* has the job of getting information and delivering it to class *B*. How do you do it?

One approach might be to have class *B* periodically check a Boolean property in class *A* named *GotNewInformation*. If *GotNewInformation* is true, then class *B* can call the method in *A* named *GetInformation*. This is a technique known as *polling*, and it's considered rather wasteful in terms of resources.

A better approach would be for class *A* to call a particular method in *B*, named perhaps *TheNewStuffIsReady*. But that means that class *B* is required to have a method named *TheNewStuffIsReady* whenever it needs to interact with class *A*. Perhaps there are other activities *B* needs to do with *A* that don't involve this transfer of information.

A mechanism for dealing with scenarios such as these is built into .NET and is known as the *event*. The event is a type-safe mechanism essentially for defining call-back functions. It is considered type-safe because the call-back function must have a specific signature defined by a *delegate*.

In this scenario, class *A* would define an event, and class *B* would define a method to function as an event handler. Class *B* registers this event handler with class *A*'s event, and then class *A* effectively calls that event handler whenever it has new information that might be of interest to class *B*.

Let's look at the different parts: Class *A* would define a public *event* member as simply as this:

```
public event EventHandler InformationAlert;
```

The name of this event is *InformationAlert*. It is associated with a delegate named *EventHandler*, which is defined in the .NET Framework. Look in the *System* namespace, and you'll see *EventHandler* defined like so:

```
public delegate void EventHandler(Object sender, EventArgs e);
```

EventArgs is a class defined in the *System* namespace, and it is the base class for many derived classes that are used in connection with events.

The delegate defines a signature for an event handler that is associated with the *InformationAlert* event. Class *B* declares a method to function as an event handler like this:

```
void MyInformationAlertHandler(object sender, EventArgs e)
{
    // process the event
}
```

Class *B* can name this event handler whatever it wants, but it must have the same return type (*void*, in this case) and the same parameters as the *EventHandler* delegate. Although naming the *EventArgs* parameter *e* has become common, you can really name it whatever you want. (I prefer *args* myself.)

Now *B* has a method that is suitable for handling event notifications from *A*. Class *B* must also register the event handler with *A*. I'll assume here that *a* is an instance of the *A* class created by class *B*. *B* "installs" or "registers" the event handler with a special syntax like this:

```
a.InformationAlert += new EventHandler(MyInformationAlertHandler);
```

The instance of the class that defined the event is followed by a period and the event name, and then the compound assignment operator, followed by the delegate constructor and the name of the event handler.

Now, whenever class *A* has new information, it "raises" or "fires" the *InformationAlert* event with code like this:

```
if (InformationAlert != null)
    InformationAlert(this, new EventArgs());
```

InformationAlert will be *null* if there aren't any event handlers registered with this event. If there are registered event handlers (and there could be more than one), class *A* effectively calls all those event handlers with the statement that follows. The two parameters that follow *InformationAlert* become the two parameters to the event handlers. The first is the object firing the event, and the second is an instance of type *EventArgs*. (If more information must be delivered to the event handler, then a class that derives from *EventArgs* would be used instead, and a different delegate would be associated with the event.)

When class *A* fires the *InformationAlert* event, then the *MyInformation-AlertHandler* method in class *B* is called.

At any time, class *B* can unregister the event handler using code like this:

```
a.InformationAlert -= new EventHandler(MyInformationAlertHandler);
```

Notice this time that the compound assignment operator is for subtraction.

In .NET 2.0, the syntax for registering and unregistering event handlers was simplified somewhat. Rather than registering an event handler like this

```
a.InformationAlert += new EventHandler(MyInformationAlertHandler);
```

you can now just use the method name:

```
a.InformationAlert += MyInformationAlertHandler;
```

Of course, *MyInformationAlertHandler* must still be defined in accordance with the *EventHandler* delegate.

Events aren't used much in character-mode programming, but they're used extensively in graphical interfaces. The keyboard, the mouse, all types of controls and menus—everything generates events, and event handling is one of the necessary skills in programming for graphical interfaces.

Regardless, let's try to put an event handler in a character-mode program. The class I've chosen is the *Timer* class from the *System.Timers* namespace. The *Timer* class serves to periodically notify a class when a period time of elapsed. It does this with an event handler defined like so:

```
public event ElapsedEventHandler Elapsed;
```

The event is named *Elapsed*, and it is associated with a delegate named *ElapsedEventHandler* and defined in the *System.Timers* namespace like so:

```
public delegate void ElapsedEventHandler(Object sender,
                                         ElapsedEventArgs e)
```

The *ElapsedEventArgs* class is also defined in *System.Timers*, and defines a *SignalTime* property, which is the *DateTime* object when the event was raised.

Here's a program that creates an object of type *Timer* and installs an event handler to be notified every second. This program requires a reference to the System.dll assembly.

SetTimer.cs

```
//---------------------------------------
// SetTimer.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.Timers;

class SetTimer
{
    static void Main()
    {
        Timer tmr = new Timer();
        tmr.Elapsed += TimerTickHandler;
        tmr.Interval = 1000;
        tmr.Enabled = true;

        Console.ReadLine();
        tmr.Elapsed -= TimerTickHandler;
    }
```

```
    static void TimerTickHandler(object sender, ElapsedEventArgs args)
    {
        Console.Write("\r{0} ", args.SignalTime.ToLongTimeString());
    }
}
```

The *Main* method creates an object of type *Timer*, and installs a handler for the *Elapsed* event. The *TimerTickHandler* method is defined in accordance with the *ElapsedEventHandler* delegate. The *Interval* is set for 1 second, and the timer is enabled.

At this point, *TimerTickHandler* is called every second by the *Timer* object. If this method needed to, it could get access to the *Timer* object raising the event by casting the *sender* parameter to a *Timer* object. But this handler just uses the *SignalTime* property of the *ElapsedEventArgs* parameter to display the current time. The use of a carriage return makes the new time overwrite the old time.

When the user presses the Enter key, the *Console.ReadLine* method returns and the program terminates. Uninstalling the event handler is not strictly necessary here, but it can't hurt.

Beginning in .NET 2.0, it is possible to define anonymous methods for use with events. Rather than declaring a method for the event handler, you put the event handling code in the statement where you normally install the handler. Here's a demonstration of this technique.

SetTimerWithAnonymousMethod.cs

```
//-----------------------------------------------------------
// SetTimerWithAnonymousMethod.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------------
using System;
using System.Timers;

class SetTimer
{
    static void Main()
    {
        Timer tmr = new Timer();
        tmr.Elapsed += delegate(object sender, ElapsedEventArgs args)
        {
            Console.Write("\r{0} ", args.SignalTime.ToLongTimeString());
        };
        tmr.Interval = 1000;
        tmr.Enabled = true;

        Console.ReadLine();
    }
}
```

The compound assignment statement is followed by the keyword *delegate* and the parameters of the delegate in parentheses. Curly brackets delimit the body of the anonymous method.

If you don't need the parameters to the event handler, you can simplify this even more:

```
tmr.Elapsed += delegate
{
    Console.Write("\r{0} ", DateTime.Now.ToLongTimeString());
};
```

Now the code uses the static *DateTime.Now* property to obtain the current time rather than the *SignalTime* property of *ElapsedEventArgs*, so the two parameters to the method can be eliminated.

Although anonymous methods are sometimes convenient, they're not the most aesthetically attractive feature of the C# language and can be tricky to use when they access local variables of the method in which they appear.

Chapter 25. Files and Streams

The *System.IO* namespace provides essential support for file input and output of both binary files and text files. However, if those text files you need to read and write are actually XML files, then you'll probably be happier using higher-level classes in the *System.Xml* namespace.

For the C programmer whose main arsenal of file I/O tools consists of library functions such as *fopen*, *fread*, *fwrite*, and *fprintf*, the *System.IO* namespace can seem excessively convoluted and complex. This chapter is intended to guide you through *System.IO* so that you can get a sense of the important classes and the rationale for its complexity. There are some shortcuts for reading entire files, but I won't show them to you until the very end of the chapter.

This chapter takes a rather roundabout path through *System.IO* that might not at first seem to be entirely rational, so let me summarize here:

The first file I/O class I'll describe is *FileStream*, which lets you read and write bytes or arrays of bytes. This is the lowest level file I/O you'll probably want to perform.

Next I'll cover classes that read and write text files and streams. These classes are *StreamReader* and *StreamWriter*, and some related files.

Then I'll get back to binary files with *BinaryReader* and *BinaryWriter*. I discuss these classes after the text I/O classes because they incorporate reading and writing strings as well as other C# data types.

Then it's time to attack the file system and directories with classes like *Environment*, *Path*, *DirectoryInfo*, *Directory*, *FileInfo*, and *File*.

Finally, I'll show how the *File* class offers some higher-level methods for working with whole files.

Overview of Streams

The .NET Framework distinguishes between files and streams. A *file* is a collection of data stored on a disk with a name and (often) a directory path. When you open a file for reading or writing, it becomes a *stream*. A stream is something on which you can perform read and write operations. But streams encompass more than just open disk files. Data coming over a network is a stream, and you can also create a stream in memory. In a console application, keyboard input and text output are also streams.

Stream is an abstract class defined in the *System.IO* namespace. This class hierarchy shows the most important classes that derive from *Stream*.

Object
MarshalByRefObject
Stream (abstract)
BufferedStream
FileStream
MemoryStream
NetworkStream

In general, a stream is an object that lets you read bytes, write bytes, and seek to a particular location. However, not all streams let you perform all these operations. The *Stream* class defines four bool properties named *CanRead*, *CanWrite*, *CanSeek*, and *CanTimeout*.

If *CanRead* is *true*, you can call *ReadByte* on the *Stream* object to read a single byte, or *Read* to read multiple bytes into a *byte* array.

If *CanWrite* is *true*, you can call *WriteByte* to write a single byte to the stream, or *Write* to write multiple bytes from a *byte* array. The *Flush* method writes any buffered output to the stream.

If *CanSeek* is *true* (which is the case for a stream based on a file) you can use the *Length* property to obtain the length of the stream, and you can use the *Position* property to obtain the current position or set the current position. Both properties are of type *long*. You can also use the *Seek* method with an offset combined with a member of the *SeekOrigin* enumeration (*Begin*, *Current*, or *End*).

If both *CanWrite* and *CanSeek* are *true*, you can call the *SetLength* method to set a new length of the stream.

If *CanTimeout* is *true* (which can be the case for a network stream), then the *ReadTimeout* and *WriteTimeout* properties let you obtain or specify the timeout values.

You can use the *BeginRead*, *EndRead*, *BeginWrite*, and *EndWrite* methods to read or write the stream asynchronously.

The *Close* method closes the stream.

The *FileStream* Class

The *FileStream* class derives from *Stream* and performs the most rudimentary file I/O. If you want to restrict yourself to learning just one class in the *System.IO* namespace, this is the one you'll need.

To open an existing file, or create a new file, you create an object of type *FileStream* using one of the constructors that begins with a *string* argument indicating the filename. Other constructors let you open files

based on operating system file handles; these are useful for interfacing with existing code.

For pure .NET programs, however, the simplest constructor requires two arguments, which results in opening a file for both reading and writing. The first argument is the filename, and the second argument is a member of the *FileMode* enumeration:

Member	Value	Caveats
FileMode.CreateNew	1	Fails if file exists.
FileMode.Create	2	Delete file contents if file already exists
FileMode.Open	3	Fails if file does not exist
FileMode.OpenOrCreate	4	Creates new file if file does not exist
FileMode.Truncate	5	Fails if file does not exist; deletes contents of file
FileMode.Append	6	Fails if file is opened for reading; creates new file if file does not exist; seeks to end of file

The constructor fails by raising an exception such as *IOException* or *FileNotFoundException*. Almost always you should call the *FileStream* constructor in a *try* block to gracefully recover from any problems regarding the presumed existence or non-existence of the file.

In addition to the required two constructor arguments, you can supply a third argument, which is a member of the *FileAccess* enumeration:

Member	Value	Description
FileAccess.Read	1	Fails for *FileMode.CreateNew*, *FileMode.Create*, *FileMode.Truncate*, or *FileMode.Append*
FileAccess.Write	2	Fails if file is read-only
FileAccess.ReadWrite	3	Fails for *FileMode.Append* or if file is read-only

There's only one case where a *FileAccess* argument is required: when you open a file with *FileMode.Append*, the constructor fails if the file is opened for reading. Because files are opened for reading and writing by default, the following constructor always fails:

```
new FileStream(strFileName, FileMode.Append)
```

To use *FileMode.Append*, you must include an argument of *FileAccess.Write*:

```
new FileStream(strFileName, FileMode.Append, FileAccess.Write)
```

Unless you specify a *FileShare* argument, the file is open for exclusive use by your process. No other process (or the same process) can open the same file. Moreover, if any other process already has the file open and you don't specify a *FileShare* argument, the *FileStream* constructor will fail. The *FileShare* argument lets you be more specific about the sharing:

Member	Value	Description
FileShare.None	0	Allow other processes no access to the file; default
FileShare.Read	1	Allow other processes to read the file
FileShare.Write	2	Allow other processes to write to the file
FileShare.ReadWrite	3	Allow other processes full access to the file

When you only need to read from a file, it's common to allow other processes to read from it as well; in other words, *FileAccess.Read* should usually be accompanied by *FileShare.Read*. This courtesy goes both ways: if another process has a file open with *FileAccess.Read* and *FileShare.Read*, your process won't be able to open it unless you specify both flags as well. The *FileStream* class defines *Lock* and *Unlock* methods for accessing shared files.

Once you open a file using one of the *FileStream* constructors, you have access to the properties and methods defined by the *Stream* class that I described above. The *CanRead* and *CanWrite* properties will depend on the *FileAccess* value you specified.

The *CanSeek* property is always *true* for open files, which means that the *Length* and *Position* properties are valid. The *Length* property is read-only; the *Position* property is read/write. Both properties are of type *long*, which means they allow file sizes of up to 9 terabytes (9×10^9 bytes).

You can set the *Position* property to seek to any point in the file. For example, you can seek to the 100th byte in the file

```
fs.Position = 100;
```

You can seek to the end of the file (for appending, perhaps)

```
fs.Position = fs.Length;
```

The *Seek* method is similar to the file-seeking functions in C. The *SeekOrigin* enumeration (with values of *Begin*, *Current*, and *End*) indicate where the offset argument is measured from.

You can read individual bytes with *ReadByte* or multiple bytes into an array with *Read*. Both return *int* values, but with different meanings: *ReadByte* normally returns the next byte from the file cast to an *int* without sign extension. For example, the bytes 0xFF becomes the integer

0x000000FF or 255. A return value of –1 indicates an attempt to read past the end of the file.

The *Read* method requires an array of type *byte*:

```
byte[] buffer = new byte[1000];
fs.Read(buffer, 0, buffer.Length);
```

The second argument is an offset into the buffer, and the third argument is the number of bytes to read. *Read* returns the number of bytes actually read, which for files is the same as the third argument to *Read* unless it's gotten to the end of the file. A return value of 0 indicates that there are no more bytes to be read. For other types of streams, *Read* can return a value less than the third argument, but always at least 1 unless the entire stream has been read.

The *WriteByte* and *Write* methods are similar.

Despite what may or may not happen as a result of garbage collection on the *FileStream* object, you should always explicitly call the *Close* method for any files you open.

FileStream is an excellent choice for a traditional hex-dump program.

HexDump.cs

```
//---------------------------------------
// HexDump.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.IO;

class HexDump
{
    public static int Main(string[] strArgs)
    {
        if (strArgs.Length > 0)
        {
            foreach (string strFileName in strArgs)
                DumpFile(strFileName);
        }
        else
        {
            string strFileName;

            do
            {
                Console.Write("Enter filename (or Enter to end): ");
                strFileName = Console.ReadLine();
                if (strFileName.Length > 0)
                    DumpFile(strFileName);
            }
            while (strFileName.Length > 0);
        }
        return 0;
    }
```

```
    protected static void DumpFile(string strFileName)
    {
        FileStream fs;

        try
        {
            fs = new FileStream(strFileName, FileMode.Open,
                                FileAccess.Read, FileShare.Read);
        }
        catch (Exception exc)
        {
            Console.WriteLine("HexDump: {0}", exc.Message);
            return;
        }
        Console.WriteLine(strFileName);
        DumpStream(fs);
        fs.Close();
    }
    static void DumpStream(Stream stream)
    {
        byte[] buffer = new byte[16];
        long addr = 0;
        int count;

        while ((count = stream.Read(buffer, 0, 16)) > 0)
        {
            Console.WriteLine(ComposeLine(addr, buffer, count));
            addr += 16;
        }
    }
    static string ComposeLine(long addr, byte[] buffer, int count)
    {
        string str = String.Format("{0:X4}-{1:X4}  ",
                            (uint)addr / 65536, (ushort)addr);

        for (int i = 0; i < 16; i++)
        {
            str += (i < count) ?
                           String.Format("{0:X2}", buffer[i]) : "  ";
            str += (i == 7 && count > 7) ? "-" : " ";
        }
        str += " ";

        for (int i = 0; i < 16; i++)
        {
            char ch = (i < count) ? Convert.ToChar(buffer[i]) : ' ';
            str += Char.IsControl(ch) ? "." : ch.ToString();
        }
        return str;
    }
}
```

This program uses the version of *Main* that has a single argument, which is an array of strings, each of which is a command-line argument to the program. Unlike the *Main* function in C, the *Main* method in C# doesn't

include an argument count and also doesn't include the program name among the arguments. Wildcards are not automatically expanded. (I'll get to that job later in this chapter.) If there are no arguments, then the program requests a filename. If you're running the program from Visual Studio, then hexdump.exe will work as an argument.

Once HexDump successfully opens each file, the program uses the *Read* method to read 16-byte chunks from the file, and then HexDump's *ComposeLine* method displays them.

Although *FileStream* is the most essential class in *System.IO* for reading and writing files, for most cases it will prove to be inadequate. The problem is that C# is not nearly as flexible as C in casting. For example, a C programmer might read a series of bytes into an array, and then use pointers into this array to cast them into other data types. You can't do this in C#, and you'd probably need to manually assemble other data types from their constituent bytes.

So, unless reading and writing arrays of bytes is entirely satisfactory to you, you probably can't limit your knowledge of file I/O to the *FileStream* class. You'll probably use *StreamReader* and *StreamWriter* for reading and writing text files, and *BinaryReader* and *BinaryWriter* for reading and writing binary files of types other than byte arrays.

Text File I/O

Certainly one important type of file is the text file, which consists entirely of lines of text separated by end-of-line markers. The *System.IO* namespace has specific classes to read and write text files. Here's the class hierarchy:

Object
- *MarshalByRefObject*
  - *TextReader* (abstract)
    - *StreamReader*
    - *StringReader*
  - *TextWriter* (abstract)
    - *StreamWriter*
    - *StringWriter*
    - …

Although these classes are not descended from *Stream*, they certainly make use of the *Stream* class.

The two classes I'm going to focus on here are *StreamReader* and *StreamWriter*, which are designed for reading and writing text files or text streams. The two other non-abstract classes are *StringReader* and *StringWriter*, which are not strictly file I/O classes, but use similar methods to read to and write from strings. I'll discuss these classes towards the end of the next chapter.

Text may seem to be a very simple form of data storage, but in recent years text has assumed a layer of complexity as a result of the increased use of Unicode. Although the C# *char* and *string* data types store text as Unicode characters, in most cases you probably do *not* want to write text as 16-bit characters, particularly if programs reading the files are expecting to encounter ASCII or one of the more efficient encodings of Unicode.

Fortunately, the *StreamWriter* class lets you have control over how the Unicode strings in your C# program are converted for storage in a file. You assert this control via classes defined in the *System.Text* namespace. Similarly, *StreamReader* lets your program read text files in various formats and convert the text from the files to Unicode strings in your program.

Let's look at *StreamWriter* first. You use this class to write to new or existing text files. *StreamWriter* has four constructors that let you specify a filename:

```
new StreamWriter(string filename)
new StreamWriter(string filename, bool append)
new StreamWriter(string filename, bool append, Encoding enc)
new StreamWriter(string filename, bool append, Encoding enc, int size)
```

These constructors open a file for writing, probably using a *FileStream* constructor. By default, if the file exists its contents will be destroyed. Set the second argument to *true* to avoid that. The *size* argument is a buffer size.

Three other *StreamWriter* constructors use an existing *Stream* object:

```
new StreamWriter(Stream strm)
new StreamWriter(Stream strm, Encoding enc)
new StreamWriter(Stream strm, Encoding enc, int size)
```

If you use a constructor without an *Encoding* argument, the resultant *StreamWriter* object will not store strings to the file in a Unicode format with two bytes per character. Nor will it convert your strings to ASCII. Instead, the *StreamWriter* uses a popular encoding format known as UTF-8.

If you want to specify an *Encoding* argument, you need an object of type *Encoding*, which is a class defined in the *System.Text* namespace. It's easiest (and in most cases, sufficient) to use one of the static properties of the *Encoding* class to obtain this object:

```
Encoding.Default
Encoding.Unicode
Encoding.BigEndianUnicode
Encoding.UTF8
Encoding.UTF7
Encoding.ASCII
```

The *Encoding* argument to *StreamWriter* can also be an instance of one of the classes in *System.Text* that derive from *Encoding*, which are *ASCIIEncoding*, *UnicodeEncoding*, *UTF7Encoding*, and *UTF8Encoding*. The constructors for these classes often have a few options, so you may want to check them out if the static properties aren't doing precisely what you want.

When you specify an encoding of *Encoding.Unicode*, each character is written to the file in two bytes with the least significant byte first, in accordance with the so-called little-endian architecture of Intel microprocessors. The file or stream begins with the bytes 0xFF and 0xFE, which correspond to the Unicode character 0xFEFF, which is defined in the Unicode standard as the byte order mark (BOM).

An encoding of *Encoding.BigEndianUnicode* stores the most significant bytes of each character first. The file or stream begins with the bytes 0xFE and 0xFF, which also correspond to the Unicode character 0xFEFF. The Unicode character 0xFFFE is intentially undefined so that applications can determine the byte ordering of a Unicode file from its first two bytes.

(Readers unsure whether little-endian or big-endian microprocessor architectures are superior should consult Jonathan Swift's *Gulliver's Travels*, Part I, Chapter 4.)

If you want to store strings in Unicode but you don't want the byte order marks emitted, you can instead obtain an *Encoding* argument for the *StreamWriter* constructor by creating an object of type *UnicodeEncoding*:

```
new UnicodeEncoding(isBigEndian, includeByteOrderMark)
```

Set the two Boolean arguments accordingly.

UTF-8 is a character encoding designed to represent Unicode characters without using any zero bytes (and hence, to be C and UNIX friendly). UTF stands for *UCS Transformation Format*. UCS stands for *Universal Character Set*, which is another name for ISO 10646, a character-encoding standard with which Unicode is compatible.

In UTF-8, each Unicode character is translated to a sequence of 1 to 6 non-zero bytes. Unicode characters in the ASCII range (0x000 through 0x007F) are translated directly to single-byte values. Thus, Unicode strings that contain only ASCII are translated to ASCII files. UTF-8 is documented in RFC 2279. (RFC stands for Request for Comments. RFCs are documentations of Internet standards and are obtainable from many sources, including the Web site of the Internet Engineering Task Force, *http://www.ietf.org*.)

When you specify *Encoding.UTF8*, the *StreamWriter* class converts the Unicode text strings to UTF-8. In addition, it writes the three bytes 0xEF,

0xBB, and 0xBF to the beginning of the file or stream. These bytes are the Unicode BOM converted to UTF-8.

If you want to use UTF-8 encoding but you don't want those three bytes emitted, don't use *Encoding.UTF8*. Use *Encoding.Default* instead or one of the constructors that don't have an *Encoding* argument. These options also provide UTF-8 encoding, but the three identification bytes are not emitted.

Alternatively, you can create an object of type *UTF8Encoding* and pass that object as the argument to *StreamWriter*. Use

```
new UTF8Encoding()
```

or

```
new UTF8Encoding(false)
```

to suppress the three bytes, and use

```
new UTF8Encoding(true)
```

to emit the identification bytes.

UTF-7 is documented in RFC 2152. Unicode characters are translated to a sequence of bytes that always have a high bit of 0. UTF-7 is intended for environments in which only 7-bit values can be used, such as e-mail. Use *Encoding.UTF7* in the *StreamWriter* constructor for UTF-7 encoding. No identification bytes are involved with UTF-7.

When you specify an encoding of *Encoding.ASCII*, the resultant file or stream contains only ASCII characters, that is, characters in the range 0x00 through 0x7F. Any Unicode character not in this range is converted to a question mark (ASCII code 0x3F). This is the only encoding in which data is actually lost.

Another important option is a text file that contains only characters from the Windows ANSI character set (characters 0x00 through 0xFF) in a one-byte-per-character format. You can't use *Encoding.ASCII* because characters 0x80 through 0xFF will be replaced by question marks. And you can't use *Encoding.UTF8* because characters 0x80 through 0xFF will be written to the file as a pair of bytes. In such a case you need to obtain an *Encoding* object using the static *GetEncoding* method of the *Encoding* class with an argument of 1252 (the code page identifier for the Windows character set). This is the argument you pass to the *StreamWriter* constructor:

```
Encoding.GetEncoding(1252)
```

The *StreamWriter* class has a few handy properties. The get-only *BaseStream* property returns either the *Stream* object you used to create the *StreamWriter* object, or the *Stream* object that the *StreamWriter* class created based on the filename you supplied. If the base stream supports

seeking, you can use that object to perform seeking operations on the stream.

The *Encoding* property of *StreamWriter* is the *Encoding* object you specified in the constructor, or a *UTF8Encoding* object otherwise. Setting the *AutoFlush* property to *true* performs a flush of the buffer after every write.

The *NewLine* property is inherited from *TextWriter*. By default, it's the string "\r\n" (carriage return and line feed), but you can change it to "\n" instead. Anything else, and the files won't be properly readable by *StreamReader* objects.

The versatility of the *StreamWriter* class becomes apparent when you look at the multitude of *Write* and *WriteLine* methods that the class inherits from *TextWriter*. These methods parallel those in the *Console* class but instead write text to a file. The *WriteLine* methods end with writing a newline character; the *Write* methods do not. Also included are versions with formatting strings. The *StreamWriter* method also inherits *Flush* and *Close* methods from *TextWriter*.

Here's a little program that appends text to the same file every time you run the program. You'll find the file in the same directory as the StreamWriterDemo.exe file.

StreamWriterDemo.cs

```
//-----------------------------------------------
// StreamWriterDemo.csn (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.IO;

class StreamWriterDemo
{
     public static void Main()
     {
          StreamWriter sw = new StreamWriter("StreamWriterDemo.txt", true);

          sw.WriteLine("You ran the StreamWriterDemo program on {0}",
                       DateTime.Now);

          sw.Close();
     }
}
```

Notice the *true* argument to the constructor, indicating that the existing file will be appended. The Unicode string in the *WriteLine* statement are converted to UTF-8, but the program doesn't use any non-ASCII characters, so the file will appear to be ASCII.

The *StreamReader* class is for reading text files or streams. Here are file constructors for opening a text file for reading:

```
new StreamWriter(string filename);
new StreamWriter(string filename, Encoding enc);
new StreamWriter(string filename, bool detect);
new StreamWriter(string filename, Encoding enc, bool detect);
new StreamWriter(string filename, Encoding enc, bool detect, int size);
```

An additional set of five constructors create a *StreamReader* object based on an existing stream:

```
new StreamWriter(Stream strm);
new StreamWriter(Stream strm, Encoding enc);
new StreamWriter(Stream strm, bool detect);
new StreamWriter(Stream strm, Encoding enc, bool detect);
new StreamWriter(Stream strm, Encoding enc, bool detect, int size);
```

If you set the *detect* argument to *true*, the constructor will attempt to determine the encoding of the file from the first two or three bytes. Or you can specify the encoding explicitly. If you set *detect* to *true* and also specify an encoding, the constructor will use the specified encoding only if it can't detect the encoding of the file. (For example, ASCII and UTF-7 can't be differentiated by inspection because they don't begin with a BOM and both contain only bytes in the range 0x00 to 0x7F.)

The *StreamReader* class contains get-only properties *BaseStream* and *CurrentEncoding*. The latter property may change between the time the object is constructed and the first read operation performed on the file because the object obtains knowledge of the identification bytes only after the first read.

You can read a text file character-by-character using the *Peek* and *Read* methods defined by *StreamReader*. Both return the next character in the stream or –1 if the end of the stream has been reached. You must explicitly cast the return value to a *char* if the return value is not –1. Or, you can read multiple characters into an array of type *char*. This *Read* overload returns the number of characters read into the array or 0 if the end of the stream has been reached.

It is more common with text files to read entire lines rather than individual characters. The *ReadLine* method reads the next line up to the next end-of-line marker, and strips the end-of-line characters from the resultant string. The method returns a zero-length character string if the line of text contains only an end-of-line marker; the method returns *null* if the end of the stream has been reached.

ReadToEnd returns everything from the current position to the end of the file. The method returns *null* if the end of the stream has been reached.

Here's a program that asks you for a URI of an HTML file (or other text file) on the Web. (An HTTP prefix must be included.) It obtains a *Stream* for that file using some boilerplate code involving the *WebRequest* and *WebResponse* classes. It then constructs a *StreamReader* object from

that stream, uses *ReadLine* to read each line, and then displays each line using *Console.WriteLine* with a line number.

HtmlDump.cs

```
//---------------------------------------
// HtmlDump.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.IO;
using System.Net;

class HtmlDump
{
    public static void Main()
    {
        Console.Write("Enter a URI: ");
        string strUri = Console.ReadLine();

        if (strUri.Length == 0)
            return;

        WebRequest webreq;
        WebResponse webres;

        try
        {
            webreq = WebRequest.Create(strUri);
            webres = webreq.GetResponse();
        }
        catch (Exception exc)
        {
            Console.WriteLine("HtmlDump: {0}", exc.Message);
            return;
        }

        if (webres.ContentType.Substring(0, 4) != "text")
        {
            Console.WriteLine("HtmlDump: URI must be a text type.");
            return;
        }

        Stream stream = webres.GetResponseStream();
        StreamReader strrdr = new StreamReader(stream);
        string strLine;
        int line = 1;

        while ((strLine = strrdr.ReadLine()) != null)
            Console.WriteLine("{0:D5}: {1}", line++, strLine);

        stream.Close();
        return;
    }
}
```

Binary File I/O

Any file that's not a text file is a binary file. I've already discussed the *FileStream* class, which lets you read and write files in bytes and byte arrays. But most binary files consist of other data types. Unless you want to write code that constructs and deconstructs integers and other types from their constituent bytes, you'll want to take advantage of the *BinaryReader* and *BinaryWriter* classes, both of which are derived from *Object*:

Object
    BinaryReader
    BinaryWriter

For both classes, the constructors require a *Stream* object. (This is later available from the get-only *BaseStream* property.) Optionally, you can also include an *Encoding* argument in the constructor for use if the file contains embedded strings.

The *BinaryWriter* class includes 18 overloads of the *Write* method. Sixteen of these overloads have just one argument, which is an object of type *bool*, *byte*, *sbyte*, *byte[]*, *char*, *char[]*, *string*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, or *decimal*.

These *Write* methods do not store any information about the type of the data. Each type uses as many bytes as necessary. For example, a *float* is stored in 4 bytes. A *bool* requires 1 byte. The sizes of arrays are not stored. A 256-element *byte* array is stored in 256 bytes.

Strings stored in the file are preceded by the byte length stored as a 7-bit encoded integer. (The 7-bit integer encoding uses as many bytes as necessary to store an integer in 7-bit chunks. The first byte of storage is the lowest 7 bits of the integer, and so forth. The high bit of each byte is 1 if there are more bytes. The *BinaryWriter* class includes a protected method named *Write7BitEncodedInt* that performs this encoding.)

In addition, two *Write* methods let you write multiple bytes or characters from a *byte* or *char* array. The *BinaryWriter* class includes a *Seek* method, a *Flush* method, and a *Close* method that closes the underlying stream that the *BinaryWriter* is based on.

The *BinaryReader* class has individual methods to read all the various types: *ReadBoolean*, *ReadByte*, *ReadBytes* (for an array), *ReadSByte*, and so forth. These methods throw an exception of type *EndOfStream-Exception* if the end of the stream has been reached.

In most cases, your program will have an intimate knowledge of a binary form it's accessing, so it can normally avoid end-of-stream conditions. However, for maximum protection, you should put your read statements in *try* blocks in case you encounter a corrupted file.

The *PeekChar* and *Read* methods return the next *char* in the file. UTF-8 encoding is assumed if you don't specify an encoding in the constructor. The methods return –1 if the end of the stream has been reached.

The Environment Class

The *Environment* class in the *System* namespace has a method named *GetLogicalDrives* that returns a *string* array of all the drives on the user's system in the form "A:\", "C:\", and so forth. In .NET 2.0, this method became pretty much obsolete with the introduction of the *DriveInfo* class in the *System.IO* namespace.

You can create an object of type *DriveInfo* using a constructor whose argument is a drive letter:

```
DriveInfo info = new DriveInfo("C");
```

However, you'll probably have more use for the static *GetDrives* method that returns an array of *DriveInfo* objects, one for each drive on the system:

```
DriveInfo[] infos = DriveInfo.GetDrives();
```

The *DriveType* property of *DriveInfo* is a member of the *DriveType* enumeration, which has members like *Removable*, *Fixed*, and *CDRom*. The *DriveInfo* class has other properties, most of which are demonstrated by this little program that displays information about all the drives on your system.

GetMyDrives.cs

```
//-----------------------------------------
// GetMyDrives.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;
using System.IO;

class GetMyDrives
{
    static void Main()
    {
        DriveInfo[] infos = DriveInfo.GetDrives();

        foreach (DriveInfo info in infos)
        {
            Console.Write("{0} {1}, ", info.Name, info.DriveType);

            if (info.IsReady)
                Console.WriteLine("Label: {0}, Format: {1}, Size: {2:N0}",
                                  info.VolumeLabel, info.DriveFormat,
                                  info.TotalSize);
            else
                Console.WriteLine("Not ready");
        }
    }
```

```
}
```

Notice that the program checks the *IsReady* property before attempting to display information that requires the system to access the drive. Without that check, some of the properties could raise exceptions if the drive is not ready. On the system I'm using as I write this, the program displays the following information:

```
A:\ Removable, Not ready
C:\ Fixed, Label: Windows XP Pro, Format: NTFS, Size: 52,427,898,880
D:\ Fixed, Label: Available, Format: NTFS, Size: 52,427,898,880
E:\ Removable, Not ready
F:\ CDRom, Not ready
G:\ CDRom, Not ready
H:\ Fixed, Label: Windows Vista, Format: NTFS, Size: 32,570,863,616
I:\ Removable, Label: BOOKS, Format: FAT, Size: 1,041,989,632
```

The Fixed drives are all partitions on a single hard drive. The second CDRom drive is actually a DVD drive, although *DriveInfo* doesn't distinguish that. The A drive is a largely unused floppy drive, of course; the E drive is a built-in Iomega Zip drive (also largely unused these days), and I is a USB flash drive, and one of my most valuable possessions.

The static *Environment.GetFolderPath* method was demonstrated in Chapter 5 to display the current user's actual directory corresponding to the alias My Documents. The argument to *GetFolderPath* is a member of the *Environment.SpecialFolder* enumeration. For My Documents, the enumeration member is *Personal*.

This *SpecialFolder* enumeration is rather odd in that it is defined within the *Environment* class. Instead of calling *GetFolderPath* like this

```
Environment.GetFolderPath(SpecialFolder.Personal)    // Won't work!
```

you need to preface *SpecialFolder* with the class in which it's defined:

```
Environment.GetFolderPath(Environment.SpecialFolder.Personal)
```

Fortunately, this type of organization of classes and enumerations is rare.

The following program uses the static *Enum.GetValues* method to obtain an array of all the members of the *Environment.SpecialFolder* enumeration in an array. It then uses *foreach* to loop through the array and display the directory associated with each member of the enumeration.

ShowSpecialFolders.cs

```
//------------------------------------------------
// ShowSpecialFolders.cs (c) 2006 by Charles Petzold
//------------------------------------------------
using System;
```

```
class ShowSpecialFolders
{
    static void Main()
    {
        Environment.SpecialFolder[] folders = (Environment.SpecialFolder[])
                        Enum.GetValues(typeof(Environment.SpecialFolder));

        foreach (Environment.SpecialFolder folder in folders)
            Console.WriteLine("{0}: {1}", folder,
                              Environment.GetFolderPath(folder));
    }
}
```

The *Environment* class has a get-only *SystemDirectory* property that returns the same directory as *GetFolderPath* with the *Environment.-SpecialFolder.System* argument.

The static *Environment.CurrentDirectory* property lets a program obtain or set the current drive and directory for the application. When setting the directory, you can use a relative directory path, including the ".." string to indicate the parent directory. To change to the root directory of another drive, use the drive letter like so:

```
Environment.CurrentDirectory = "D:\\";
```

File and Path Name Parsing

Sometimes you need to parse and scan filenames and path names. Your program may have a fully qualified filename and you may need just the directory or the drive.

The *Path* class defined in the *System.IO* namespace consists solely of static methods and static read-only fields that ease jobs like this.

Several static methods of the *Path* class accept a path name argument in the form of a *string* and return information about it:

- *Path.IsPathRooted* tells you if the path name begins with a drive or a backslash.
- *Path.HasExtension* tells you if the filename has an extension.
- *Path.GetFileName* returns just the filename part of the file path.
- *Path.GetFileNameWithoutExtension* returns the filename without the extension.
- *Path.GetExtension* returns just the filename extension.
- *Path.GetDirectoryName* returns just the directory path of the file path.
- *Path.GetFullPath* possibly prepends the current drive and directory to the file path.
- *Path.GetPathRoot* obtains the initial drive or backslash (if any).

None of these methods require that the path or file actually exist. They're really just performing some string parsing and manipulation, possibly in combination with the current drive and directory.

The *Path.Combine* method has two arguments. The method combines a path name (the first argument) with a path and/or filename (the second argument). Use *Path.Combine* rather than string concatenation for this job. Otherwise, you have to worry about whether a backslash is at the end of the first argument or the beginning of the second.

The *Path.ChangeExtension* method also has two arguments: a path name or filename, and a new extension, including a period. Set the second argument to *null* to remove an extension.

Three methods return appropriate directories for storing temporary data. The *TempPath* returns a directory name, and both *GetTempFileName* and *GetRandomFileName* return fully qualified unique filenames.

The read-only fields of the *Path* class store characters used in file and path names.

Parallel Classes

Another common file I/O job is obtaining lists of all files and subdirectories in a particular directory. Historically, this job has always been a bit awkward. The standard libraries associated with the C programming language didn't include such a facility, probably because UNIX directory lists were text files that programs could directly access and parse.

Four classes provide you with information about files and directories: *Directory*, *File*, *DirectoryInfo*, and *FileInfo*:

Object
- *Directory*
- *File*
- *MarshalByRefObject*
  - *FileSystemInfo* (abstract)
    - *DirectoryInfo*
    - *FileInfo*

The *Directory* and *File* classes are declared as static, meaning they contain only static methods. In contrast, *DirectoryInfo* and *FileInfo* contain *no* static properties or methods. You must create an object of type *DirectoryInfo* or *FileInfo* to use these classes.

As the names suggest, both *Directory* and *DirectoryInfo* provide similar methods, except that the *Directory* methods are static and require an argument that is a directory name. The *DirectoryInfo* properties and methods are not static; the constructor argument indicates the directory name to which the properties and methods apply.

Similarly, the *File* and *FileInfo* classes provide similar methods, except that you indicate a particular filename in the static *File* methods and you create an instance of *FileInfo* by specifying a filename in the *FileInfo* constructor.

If you need information about a particular file, you may wonder whether it's best to use *File* or *FileInfo* (or similarly for directories, whether to use *Directory* or *DirectoryInfo*). If you need only one item of information, it's probably easiest to use the static classes. But if you need multiple items, it makes more sense to create an object of type *FileInfo* or *DirectoryInfo* and then use the instance properties and methods. But don't feel pressured to use one class in preference to the other.

Working with Directories

To use any of the properties or methods of the *DirectoryInfo* class, you need a *DirectoryInfo* object. One way you obtain such an object is by using the *DirectoryInfo* constructor:

```
DirectoryInfo dirinfo = new DirectoryInfo(strPath);
```

The directory doesn't have to exist. Indeed, if you want to create a new directory, creating an object of type *DirectoryInfo* is a first step.

After creating an object of type *DirectoryInfo*, you can determine whether the directory exists with the *Exists* property. Even if the directory does not exist, you can obtain information about it as if it did exist, such as the *Name* (just the subdirectory), *FullName* (the fully-qualified directory), and *Extension*, if any. The *Parent* and *Root* properties both return objects of type *DirectoryInfo*.

A few of the properties in *DirectoryInfo* are also duplicated as static methods in the *Directory* class. These are *Exists*, *GetDirectoryRoot*, and *GetParent*.

You can create a *DirectoryInfo* object based on a directory that doesn't exist. You can then create that directory on the disk by calling the *Create* method. You can also call *CreateSubdirectory* on the directory:

```
DirectoryInfo subdir = dirinfo.CreateSubdirectory(strSubPath);
```

CreateSubdirectory returns another *DirectoryInfo* object with information about the new directory. If the indicated directory already exists, no exception is thrown. The directory name used to create the *DirectoryInfo* object (or passed to the *CreateSubdirectory* method) can contain multiple levels of directory names.

If the directory doesn't exist when you create the *DirectoryInfo* object and then you call *Create*, the *Exists* property won't suddenly become *true*. You must call the *Refresh* method to refresh the *DirectoryInfo* information.

The *Directory* class also has a static method to create a new directory.

Both *Directory* and *DirectoryInfo* have methods named *Delete* to remove a directory. Both classes have overloads of *Delete* that let you specify a *bool* argument that indicates whether you want the deletion to include all files and subdirectories as well. Otherwise, the directory must be empty or an exception will be thrown.

DirectoryInfo has four read-write properties that let you obtain the *Attributes*, *CreationTime*, *LastAccessTime*, and *LastWriteTime* of the directory. (Except for *Attributes*, these properties are duplicated by static methods in the *Directory* class.) The *Attributes* property is a bitwise combination of members of the *FileAttributes* enumeration. Each member of this enumeration has a fairly familiar name like *ReadOnly*, *Hidden*, *System*, *Directory*, *Archive* and so forth, and the numerical value of each member is a power of two. You'll want to use the bitwise AND operator to test whether a bit is set. If *dirinfo* is an object of type *DirectoryInfo*, then the following expression is non-zero:

```
dirinfo.Attributes & FileAttributes.Directory
```

You can move a directory and its contents to another location on the same disk with the *MoveTo* method of *DirectoryInfo* or the static *Directory.Move* method.

The remaining methods of *Directory* and *DirectoryInfo* I want to discuss here all obtain an array of files or subdirectories in a directory, or only those files or directories that match a specified pattern using wildcards (question marks and asterisks).

The static methods of the *Directory* class all return arrays of strings. *Directory.GetDirectories* requires a directory path and an optional search pattern and returns an array of all subdirectories in the specified directory. Similarly, *Directory.GetFiles* returns an array of files in the directory. *Directory.GetFileSystemEntries* returns subdirectories and files. This last method has limited use. Because all you get is an array of strings, you probably won't be able to immediately tell which is a subdirectory and which is a file.

The *DirectoryInfo* class has similar instance methods. However, these methods do *not* return arrays of strings. The *GetDirectories* method returns an array of *DirectoryInfo* objects:

```
DirectoryInfo[] subdirs = GetDirectories();
```

Similarly, the *GetFiles* method returns an array of *FileInfo* objects, and the *GetFileSystemInfos* method returns an array of *FileSystemInfo* objects. This last method returns both subdirectories and files, but you can immediately tell which is which by examining the *FileAttributes.-Directory* bit in the *Attributes* property.

These methods allow us to enhance the HexDump program shown earlier so that it works with wildcard file specifications. Here's WildCardHexDump.

WildCardHexDump.cs

```
//-----------------------------------------
// WildCardHexDump.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.IO;

class WildCardHexDump: HexDump
{
     public new static int Main(string[] strArgs)
     {
         if (strArgs.Length > 0)
         {
             foreach (string strFileName in strArgs)
                 ExpandWildCard(strFileName);
         }
         else
         {
             string strFileName;

             do
             {
                 Console.Write("Enter filename (or Enter to end): ");
                 strFileName = Console.ReadLine();
                 if (strFileName.Length > 0)
                     ExpandWildCard(strFileName);
             }
             while (strFileName.Length > 0);
         }
         return 0;
     }
     static void ExpandWildCard(string strWildCard)
     {
          string[] strFiles;

          try
          {
               strFiles = Directory.GetFiles(strWildCard);
          }
          catch
          {
               try
               {
                    string strDir  = Path.GetDirectoryName(strWildCard);
                    string strFile = Path.GetFileName(strWildCard);

                    if (strDir == null || strDir.Length == 0)
                         strDir = ".";
```

```
                    strFiles = Directory.GetFiles(strDir, strFile);
               }
               catch
               {
                    Console.WriteLine(strWildCard + ": No Files found!");
                    return;
               }
          }
          if (strFiles.Length == 0)
               Console.WriteLine(strWildCard + ": No files found!");

          foreach(string strFile in strFiles)
               DumpFile(strFile);
     }
}
```

The *WildCardHexDump* class derives from the *HexDump* class, so the project requires a link to HexDump.cs. Because *WildCardHexDump* has its own *Main* method, the method requires a *new* keyword to avoid a warning message from the C# compiler. Also, you need to indicate to the C# compiler which *Main* method is the actual entry point to the program. You do this by bringing up the Project Properties page in Visual Studio, selecting the Application tab at the left, and in the Startup Object drop-down, selecting *WildCardHexDump*.

Besides normal wildcards, I wanted to be able to specify just a directory name as an argument. For example, I wanted C:\ to be the equivalent of C:*.*. The *ExpandWildCard* method begins by attempting to obtain all the files for the particular argument with a call to *Directory.GetFiles*. This call will work if *strWildCard* specifies only a directory. Otherwise, it throws an exception, and that's why it's in a *try* block. The *catch* block assumes that the command-line argument has path and filename components, and it obtains these components using the static *GetDirectoryName* and *GetFileName* methods of *Path*. However, the *GetFiles* method of *Directory* doesn't want a first argument that is *null* or an empty string. Before calling *GetFiles*, the program avoids that problem by setting the path name to ".", which indicates the current directory.

File Manipulation and Information

Like the *Directory* and *DirectoryInfo* classes, the *File* and *FileInfo* classes are very similar and share a great deal of functionality. Like the *Directory* class, the *File* class is static and consists entirely of static methods. The first argument to every method is a string that indicates a filename. The *FileInfo* class inherits from *FileSystemInfo*. You create an object of type *FileInfo* based on a filename that could include a full or relative directory path:

```
FileInfo fileinfo = new FileInfo(strFileName);
```

The file doesn't have to exist. You can determine whether the file exists and obtain some other information about the file using read-only properties *Exists*, *Name*, *FullName*, *Extension*, *DirectoryName* (which returns a string), *Directory* (which returns a *DirectoryInfo* object), and *Length*. Only the *Exists* property is duplicated by a method in the *File* class.

FileInfo has gettable and settable properties *Attributes*, *CreationTime*, *LastAccessTime*, and *LastWriteTime*. These properties are duplicated by static methods in the *File* class.

The *CopyTo*, *MoveTo*, and *Delete* methods of *FileInfo* are duplicated by *Copy*, *Move*, and *Delete* methods in *File*.

The *File* class also contains a collection of methods that create new files or open existing files. These methods are handy if you've obtained an array of *FileInfo* objects from a *GetFiles* call on a *DirectoryInfo* object and you want to poke your nose into each and every file.

If *fileinfo* is an object of type *FileInfo*, the following methods all return open *FileStream* objects. In these calls, *mode* is a member of the *FileMode* enumeration, *access* is a member of the *FileAccess* enumeration, and *share* is a member of the *FileShare* enumeration.

```
FileStream fstream = fileinfo.Create();
FileStream fstream = fileinfo.Open(mode);
FileStream fstream = fileinfo.Open(mode, access);
FileStream fstream = fileinfo.Open(mode, access, share);
FileStream fstream = fileinfo.OpenRead();
FileStream fstream = fileinfo.OpenWrite();
```

The following methods create or open text files:

```
StreamWriter writer = fileinfo.CreateText();
StreamWriter writer = fileinfo.AppendText();
StreamReader reader = fileinfo.OpenText();
```

The *File* class has similar static methods, each of which requires a file-name as its first argument. However, these static methods don't provide any real advantage over using the appropriate constructors of the *File-Stream*, *StreamReader*, or *StreamWriter* classes. Indeed, the very presence of these static methods in the *File* class was one of the aspects of the entire *System.IO* namespace I initially found the most confusing. It didn't (and still doesn't) make sense to use a class like *File* merely to obtain an object of type *FileStream* so that you can then use *FileStream* properties and methods. It's easier to stick to a single class if that's sufficient for your purposes.

File Class Shortcuts

However, in .NET 2.0 some additional static methods were added to the *File* class that make a whole *lot* of sense. These methods let you open, read from (or write to), and close a file, all in one statement.

Here are static methods that read entire files:

```
byte[] buffer = File.ReadAllBytes(strPathName);
string strFile = File.ReadAllText(strPathName);
string[] strLines = File.ReadAllLines(strPathName);
```

The two methods that read text files also let you specify an optional *Encoding* argument.

In the following methods, *buffer* is a *byte* array and *lines* is a *string* array.

```
File.WriteAllBytes(strPathName, buffer);
File.WriteAllText(strPathname, strFileText);
File.AppendAllText(strPathName, strAppendText);
File.WriteAllLines(strPathName, lines);
```

The three methods that involve text files let you specify an optional *Encoding* argument.

If you have a need to replace an entire file and, in the process, make the existing file into a backup copy, you'll also want to explore the *File.Replace* method, also new in .NET 2.0.

Chapter 26. String Theory

Once a C# string is created, neither the length nor the individual characters that make up the string can be changed. A C# string is thus said to be *immutable*. Whenever you need to change a string in some way, you must create another string. Many members of the *String* class create new strings based on existing strings. Many methods and properties throughout the .NET Framework create and return strings.

You may wonder if there's a performance penalty associated with frequent re-creation of *String* objects. Consider the following program, which uses the addition compound assignment operator in 10,000 string-appending operations to construct a large string.

StringAppend.cs

```
//-----------------------------------------
// StringAppend.cs (c) 2006 by Charles Petzold
//-----------------------------------------
using System;
using System.Diagnostics;

class StringAppend
{
    const int iterations = 10000;

    public static void Main()
    {
        Stopwatch watch = new Stopwatch();
        string str = String.Empty;

        watch.Start();

        for (int i = 0; i < iterations; i++)
            str += "abcdefghijklmnopqurstuvxyz\r\n";

        watch.Stop();
        Console.WriteLine(watch.ElapsedMilliseconds);
    }
}
```

The program uses the *Stopwatch* class from the *System.Diagnostics* namespace to calculate an elapsed time. The stopwatch is started before the *for* loop, stopped afterwards, and then the elapsed time in milliseconds is displayed. (The *Stopwatch* class requires a reference to the System.dll assembly.)

Each string-appending operation causes a new *string* object to be created, which requires another memory allocation. Each previous string

is marked for garbage collection. How fast this program runs depends on how fast your machine is. My pokey machine requires about 8 seconds.

A better solution in this case is the appropriately named *StringBuilder* class, defined in the *System.Text* namespace. Unlike the string maintained by the *String* class, the string maintained by *StringBuilder* can be altered. *StringBuilder* dynamically reallocates the memory used for the string. Whenever the size of the string is about to exceed the size of the memory buffer, the buffer is doubled in size. To convert a *StringBuilder* object to a *String* object, call the *ToString* method.

Here's a revised version of the program using *StringBuilder*.

StringBuilderAppend.cs

```
//-------------------------------------------------
// StringBuilderAppend.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;
using System.Diagnostics;
using System.Text;

class StringBuilderAppend
{
    const int iterations = 10000;

    public static void Main()
    {
        StringBuilder builder = new StringBuilder();
        Stopwatch watch = new Stopwatch();
        watch.Start();

        for (int i = 0; i < iterations; i++)
            builder.Append("abcdefghijklmnopqurstuvxyz\r\n");

        string str = builder.ToString();
        watch.Stop();
        Console.WriteLine(watch.ElapsedMilliseconds);
    }
}
```

You'll probably find that this program does its work a thousand times faster than the previous program. On my machine, it required 4 milliseconds.

Another efficient approach is to use the *StringWriter* class defined in the *System.IO* namespace. As I indicated in Chapter 25, both *StringWriter* and *StreamWriter* (which you use for writing to text files) derive from the abstract *TextWriter* class. Like *StringBuilder*, *StringWriter* assembles a composite string. The big advantage with *StringWriter* is that you can use the whole array of *Write* and *WriteLine* methods defined in the *TextWriter* class. Here's a sample program that performs the same task as the previous two programs but using a *StringWriter* object.

StringWriterAppend.cs

```
//-------------------------------------------------
// StringWriterAppend.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;
using System.Diagnostics;
using System.IO;

class StringWriterAppend
{
    const int iterations = 10000;

    public static void Main()
    {
        StringWriter writer = new StringWriter();
        Stopwatch watch = new Stopwatch();
        watch.Start();

        for (int i = 0; i < iterations; i++)
            writer.WriteLine("abcdefghijklmnopqurstuvxyz");

        string str = writer.ToString();
        watch.Stop();
        Console.WriteLine(watch.ElapsedMilliseconds);
    }
}
```

The speed of this program is comparable to StringBuilderAppend.

There's a lesson in all this. As operating systems, programming languages, class libraries, and frameworks provide an ever increasingly higher level of abstraction, we programmers can sometimes lose sight of all the mechanisms going on beneath the surface. What looks like a simple addition in code can actually involve many layers of low-level activity.

Chapter 27. Generics

C# generics were introduced with C# 2.0, and use a syntax that is similar to the C++ template. Generics help make classes more versatile by letting them be customized for different data types. In certain circumstances, generics preserve strong typing in places where it might otherwise have to be abandoned.

Suppose you're designing a two-dimensional graphics programming system, and you want the programmer using your system to express coordinate points in two different ways. You want integer coordinates for performance, but floating-point coordinates for precision.

You might start out designing two different classes. Here's an extremely early version of the *IntegerPoint* class:

```
class IntegerPoint
{
    public int X;
    public int Y;

    public double DistanceTo(IntegerPoint pt)
    {
        return Math.Sqrt((X - pt.X) * (X - pt.X) +
                         (Y - pt.Y) * (Y - pt.Y));
    }
}
```

Of course, you know that eventually you'll have public properties named *X* and *Y*, and private fields named, perhaps, *x* and *y*, and you'll want constructors, and probably other properties and methods. But this is how you start. Notice that the *DistanceTo* method has a parameter of another *IntegerPoint* object and calculates the distance between the two points using the Pythagorean Theorem

You also create a class named *DoublePoint*:

```
class DoublePoint
{
    public double X;
    public double Y;

    public double DistanceTo(DoublePoint pt)
    {
        return Math.Sqrt((X - pt.X) * (X - pt.X) +
                         (Y - pt.Y) * (Y - pt.Y));
    }
}
```

In this class the two fields are declared as type *double*, and the parameter to *DistanceTo* is a *DoublePoint*.

Even at this stage, you know that these two classes are going to be pretty similar except for the data types, and you'd prefer not to duplicate a lot of code. The solution is to make a generic *Point* class:

```
class Point<T>
{
    public T X;
    public T Y;

    public double DistanceTo(Point<T> pt)
    {
        ...
    }
}
```

To make a generic class, you follow the name of the class with angle brackets enclosing a placeholder that is commonly named *T* for "type," although you can name it whatever you want. This is called a *type parameter*. Notice that the two fields are now of type *T*, and the parameter to the *DistanceTo* method is an object of type *Point<T>*. However, the return value from *DistanceTo* is still a *double* because that's the return value from the *Math.Sqrt* method used for the calculation. (You can also define generic structures and generic interfaces.)

To declare a *Point* object where the coordinates are integers, you use:

```
Point<int> pti;
```

In declaring objects of the generic *Point* class you must follow the class name with an actual type in angle brackets, in this case *int*. You can also supply a *new* expression because the class has a default parameterless constructor:

```
Point<int> pti = new Point<int>();
```

The class name with the type in angle brackets is part of the *new* expression as well. In this case, the fields of the *pti* object are of type *int*, and you can assign these fields integer values:

```
pti.X = 26;
pti.Y = 14;
```

You can declare a *Point* object where the coordinates are *double* values using:

```
Point<double> ptd;
```

Now the type of the *X* and *Y* fields are *double*, and you can assign the fields *double* values:

```
ptd.X = 13.25;
ptd.Y = 3E-1;
```

You can even declare a *Point* object where the coordinates are *object* values:

```
Point<object> pto;
```

Of course, it's not clear at all what exactly this means.

I haven't yet shown you the body of the *DistanceTo* method in the generic *Point* class, because therein is a big problem. The *DistanceTo* method requires that values of *X* and *Y* be subtracted from each other. What is the type of *X* and *Y*? Well, it's whatever type goes in the angle brackets when an object of the generic *Point* class is declared. It could be *int*. It could be *double*. It could be *DateTime*. It could be *FileStream*. It could be *object*. Do all these types support the subtraction operator? No, they do not, and that's why the C# compiler will not allow you to write a *DistanceTo* method like this:

```
public double DistanceTo(Point<T> pt)    // Won't work!
{
    return Math.Sqrt((X - pt.X) * (X - pt.X) +
                     (Y - pt.Y) * (Y - pt.Y));
}
```

If *X* and *Y* can be any type, then this code is syntactically incorrect, because it cannot be executed for any arbitrary type.

Because the argument to the *Math.Sqrt* method is of type *double*, you might consider casting *X* and *Y* to type *double* in this method. That won't work either, because you can't cast an arbitrary object to *double*. But you're on the right track.

To help solve problems such as these, C# generics support *constraints*, which restrict the class to certain types. The constraints use the *where* keyword. For example, you can restrict the type parameter to *SomeBaseClass* and all classes that derive from *SomeBaseClass* with the following syntax:

```
class Point<T> where T: SomeBaseClass
```

You can restrict the type parameter to value types using

```
class Point<T> where T: struct
```

Or you can use the word *class* to restrict the type parameter to reference types. You can even require that the type parameter have a parameter-less constructor:

```
class Point<T> where T: new()
```

Or you can constrain the type parameter to a class that has a parameterless constructor:

```
class Point<T> where T: class, new()
```

None of these solve our problem. For example, there is no constraint that lets you indicate that the type supports the subtraction operator.

However, you *can* indicate that *T* implements a particular interface (or multiple interfaces), and this is the feature that's going to come to our rescue. Take a look at the *IConvertible* interface defined in the *System* namespace. Classes or structures that implement this interface must support a bunch of methods for converting to the basic types, and in particular, *ToDouble*. All the basic types—and some other types as well—implement the *IConvertible* interface.

The generic *Point* class can include an *IConvertible* constraint like so:

```
class Point<T> where T:IConvertible
```

Now the compiler knows that any object of type *T* has a method named *ToDouble*, and you can write the *DistanceTo* method like this:

```
public double DistanceTo(Point<T> pt)
{
    return Math.Sqrt(Math.Pow(X.ToDouble(fmt) - pt.X.ToDouble(fmt), 2) +
                     Math.Pow(Y.ToDouble(fmt) - pt.Y.ToDouble(fmt), 2));
}
```

The *fmt* argument to the *ToDouble* methods must be an object of a type that implements the *IFormatProvider* interface, and in this case on object of type *NumberFormatInfo* is suitable, such as *NumberFormatInfo.CurrentInfo*.

And here is the generic *Point* class.

Point.cs

```
//--------------------------------------
// Point.cs (c) 2006 by Charles Petzold
//--------------------------------------
using System;
using System.Globalization;

class Point<T> where T:IConvertible
{
    public T X;
    public T Y;
    NumberFormatInfo fmt = NumberFormatInfo.CurrentInfo;

    // Parameterless Constructor
    public Point()
    {
        X = default(T);
        Y = default(T);
    }

    // Two-Parameter Constructor
    public Point(T x, T y)
    {
        X = x;
        Y = y;
    }
```

```
    public double DistanceTo(Point<T> pt)
    {
        return Math.Sqrt(Math.Pow(X.ToDouble(fmt) - pt.X.ToDouble(fmt), 2) +
                         Math.Pow(Y.ToDouble(fmt) - pt.Y.ToDouble(fmt), 2));
    }
}
```

I've also added a parameterless constructor and a two-parameter constructor to show you what those look like. The parameterless constructor really wants to set the two fields to zero, but it cannot. Instead, it uses a *default* operator that sets value types to their zero values and reference types to *null*.

The Point.cs file is part of the GenericPoints projects, which also includes the following file to test out the generic *Point* class.

GenericPoints.cs

```
//--------------------------------------------
// GenericPoints.cs (c) 2006 by Charles Petzold
//--------------------------------------------
using System;

class GenericPoints
{
    static void Main()
    {
        // Points based on integers
        Point<int> pti1 = new Point<int>();
        Point<int> pti2 = new Point<int>(5, 3);

        Console.WriteLine(pti1.DistanceTo(pti2));

        // Points based on doubles
        Point<double> ptd1 = new Point<double>(13.5, 15);
        Point<double> ptd2 = new Point<double>(3.54, 5E-1);

        Console.WriteLine(ptd2.DistanceTo(ptd1));

        // Points based on strings
        Point<string> pts1 = new Point<string>("34", "27");
        Point<string> pts2 = new Point<string>("0", "0");

        Console.WriteLine(pts1.DistanceTo(pts2));
    }
}
```

The program creates two objects of type *Point<int>*, two objects of type *Point<double>*, and—amazingly enough—two objects of type *Point<string>*. Yes, the *String* class also implements the *IConvertible* interface, and includes a method named *ToDouble* that undoubtedly calls *Double.Parse*. You can also declare objects of type *Point<DateTime>* because *DateTime* implements *IConvertible* as well. But *Point<Object>* won't work.

Where generics had the biggest impact in the .NET Framework is with the *System.Collections* namespace. With .NET 2.0, that namespace has been largely superseded by the *System.Collection.Generic* namespace, which includes generic versions of *Queue*, *Stack*, *Dictionary*, *SortedList*, and *List* (which is the generic version of *ArrayList*). These versions provide type safety that the non-generic versions do not, and are now preferred for most applications.

For example, if you need to maintain a collection of *DateTime* objects, and you can't use an array because the number you'll eventually need cannot be determined, you can use a generic List class:

```
List<DateTime> lst = new List<DateTime>();
```

All the methods such as *Add* will require that the parameter be of type *DateTime*, and the indexer is also of type *DateTime*.

As generic classes such as Dictionary demonstrate, it is possible to have multiple types in a generic class definition:

```
public class Dictionary<TKey, TValue>
```

Dictionary implements many interfaces, including *IDictionary<TKey, TValue>*.

Chapter 28. Nullable Types

Classes are reference types and structures are value types. An instance of any class can take on a *null* value, but an instance of a structure cannot.

For many applications, this distinction (and the resultant limitation) works just fine. But sometimes it would be nice to have just a little bit more information to accompany our value types.

For example, suppose our database application calls a method named *Fish* to fish something out of a database. The item can't be found, so the *Fish* method returns *null*. The *null* value basically means "it's not there" or "I can't find it." This scheme works fine if the *Fish* method is searching for an instance of a class. But suppose the *Fish* method is looking for an instance of a value type, perhaps a *DateTime* satisfying particular criteria. The *Fish* method can't return *null* because *DateTime* is a structure. The best it can do is return some pre-defined *DateTime* value that represents the case where "it's not there," perhaps *DateTime.MinValue* or *DateTime.MaxValue*.

Another example: You're accessing some XML where a *Count* attribute is normally set to an integer. However, the *Count* attribute is optional, and its absence means that the *Count* is "not applicable" for this particular case. How do you store the value of *Count* in your program? You can't just make it an *int* because you're not taking account of the "not applicable" case. You might create a *bool* named *CountIsApplicable*, but it would be even nicer having the "not applicable" case somehow stored in the same variable as the *Count* itself.

This is the rationale behind "nullable" types, which were implemented in .NET 2.0. Any value type—*int*, *bool*, *DateTime*, or any structure that you define—can be made into a "nullable," and here's where it gets bit confusing: When a value type is made into a nullable, it actually *doesn't* mean that the object can have a *null* value. It only *seems* to have that capability when you're coding in C#. In actuality, the value type is merely associated with a *bool* that indicates if the value is present or if "it's not there." Syntactically, the "it's not there" case is treated in C# as a *null*. This will all become more evident as we probe deeper into nullable types.

Nullable types have already been put to use: In the Windows Presentation Foundation, the *IsChecked* property of the *CheckBox* control is a nullable *bool*, which means that it can be *true*, *false*, or *null*. The *null* value indicates the "indeterminate" state for a tri-state *CheckBox*.

I also found a nullable type to be convenient when I was writing my WPF book. I wanted to represent birthdates and death dates as *DateTime* objects, but I also wanted a way to indicate that someone was still alive. I used a nullable *DateTime* for the death date, where a *null* value basically means "non-applicable."

The implementation of nullable types in .NET 2.0 and C# required changes to three areas:

- A *Nullable* generic structure was added to the *System* namespace.
- C# needed to recognize nullable types in some cases.
- The CLR needed to recognize nullable types for boxing.

I will cover these changes in the order I listed them, which I think is the clearest approach, although the initial syntax may look a bit clunky.

I am not privy to the internals of .NET, but I am fairly confident that the core functionality of the *Nullable* generic structure looks something like this:

```
public struct Nullable<T> where T : struct    // Pure supposition
{
    T value;
    bool hasValue;

    // Constructor
    public Nullable(T value)
    {
        this.value = value;
        hasValue = true;
    }

    // Read-Only Properties
    public bool HasValue
    {
        get { return hasValue; }
    }
    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException(
                            "Nullable object must have a value");
            return value;
        }
    }
    ...
}
```

Notice that the underlying type is restricted to value types. The *Nullable* generic structure has a default empty parameterless constructor, of

course, as well as a parametered constructor. It has two public read-only properties named *HasValue* and *Value*.

Let's create an object of type nullable *DateTime* using the parameterless constructor:

```
Nullable<DateTime> ndt = new Nullable<DateTime>();
```

I've named this variable *ndt* to stand for "nullable *DateTime*." Because the parameterless constructor is used here, the *hasValue* field has its default value of *false*, and *HasValue* also returns *false*. Any attempt to access the *Value* parameter raises an *InvalidOperationException*.

Now let's use the other constructor. If we're creating a nullable *DateTime*, the constructor requires a *DateTime* argument:

```
Nullable<DateTime> ndt = new Nullable<DateTime>(DateTime.Now);
```

Now *ndt.HasValue* equals *true*, and *ndt.Value* returns an object of type *DateTime*. If you're writing code that must deal with an object of type nullable *DateTime*, you might write code that looks something like this:

```
if (ndt.HasValue)
    Console.WriteLine(ndt.Value.Year);
else
    Console.WriteLine("Year not available");
```

Or, you might want to extract the actual *DateTime* object from the nullable *DateTime* object and then use that:

```
if (ndt.HasValue)
{
    DateTime dt = ndt.Value;
    Console.WriteLine(dt.Year);
    ...
}
```

The *Nullable* generic structure also defines an implicit cast and an explicit cast to ease some of the syntax:

```
public struct Nullable<T> where T : struct    // Pure supposition
{
    ...

    public static implicit operator Nullable<T>(T value)
    {
        return new Nullable<T>(value);
    }
    public static explicit operator T(Nullable<T> value)
    {
        return value.Value;
    }

    ...
}
```

The implicit cast allows you to set an object of a nullable type directly from the underlying type. In the example of the nullable *DateTime*, you can do this:

```
ndt = DateTime.Now;
```

Going the other way is more problematic:

```
DateTime dt = ndt;     // Won't work!
```

There is no implicit cast for assigning a nullable *DateTime* to a *DateTime*, and it's easy to see why: The assignment won't work if *ndt.HasValue* is *false*. When assigning a nullable to a non-nullable, an explicit cast is required:

```
DateTime dt = (DateTime) ndt;
```

Now the programmer's intention is clear, and it is assumed the programmer knows what she's doing. If *ndt.HasValue* is, in fact, *false*, the statement will raise an *InvalidOperationException* when the *Value* property is accessed in code for the explicit cast.

The *Nullable* structure also has two versions of a method named *GetValueOrDefault* that may be handy in some cases. Like the *Value* property, this method returns an object of the underlying type, but it does not raise an exception if *HasValue* is *false*. Instead, it returns the default value of the underlying type.

For example:

```
DateTime dt = ndt.GetValueOrDefault();
```

If *ndt.HasValue* is *true*, the method returns *ndt.Value*. Otherwise, the method returns a new instance of *DateTime* created with a parameterless constructor, which is *DateTime.MinValue*.

An overload of the *GetValueOrDefault* method lets you specify the value returned if the object is *null*. For example:

```
DateTime dt = ndt.GetValueOrDefault(new DateTime(1900, 1, 1));
```

Now if *ndt.HasValue* is *false*, *dt* is set to the date January 1, 1900.

I suspect that *GetValueOrDefault* is implemented something like this:

```
public struct Nullable<T> where T : struct    // Pure supposition
{
    ...

    public T GetValueOrDefault()
    {
        return HasValue ? Value : new T();
    }
    public T GetValueOrDefault(T defaultValue)
    {
        return HasValue ? Value : defaultValue;
    }
```

```
    ...
}
```

You probably want to be able to pass an instance of a nullable to *ToString* and have something reasonable happen. You can do that because the *Nullable* generic structure overrides *ToString*. *GetHashCode* is also overridden:

```
public struct Nullable<T> where T : struct    // Pure supposition
{
    ...

    public override string ToString()
    {
        return HasValue ? Value.ToString() : "";
    }
    public override int GetHashCode()
    {
        return HasValue ? Value.GetHashCode() : 0;
    }

    ...
}
```

Notice that the expression

```
ndt.ToString()
```

is perfectly valid but the expression

```
ndt.ToString("D")    // No good!
```

is not. Nor is:

```
ndt.ToLongDateString()    // No good!
```

If you want to use anything but the default parameterless *ToString* method, you need to access the *Value* property:

```
ndt.Value.ToString("D")
ndt.Value.ToLongDateString()
```

The *Nullable* structure also overrides the *Equals* method. Two objects of the same nullable type are considered equal only if their *HasValue* properties are equal, and if *HasValue* is *true*, if their *Value* properties are equal:

```
public struct Nullable<T> where T : struct    // Pure supposition
{
    ...
    public override bool Equals(object obj)
    {
        if (obj.GetType() != GetType())
            return false;

        Nullable<T> nt = (Nullable<T>)obj;
```

```
            if (nt.HasValue != HasValue)
                return false;

            return HasValue && Value.Equals(nt.Value);
        }
    }
```

And that's the end of my re-creation of the generic *Nullable* structure, and at this point you might wonder why it's named *Nullable*. Have you seen any *null* keywords around? *Nullable* is a structure, and no instance of a structure can be *null*, and the underlying type of *Nullable* is also a structure.

So where does *null* come into the picture?

That's what the C# compiler adds to the equation. I don't know exactly how much C# gets involved with nullable types, but if you do some experimentation and look at the CIL, you'll see for yourself that an assignment statement like

```
Nullable<DateTime> ndt = null;
```

generates the same CIL as:

```
Nullable<DateTime> ndt = new Nullable<DateTime>();
```

The object really isn't being set to *null*! It's merely being recreated so its *HasValue* property returns *false*. Similarly, C# treats the expression

```
ndt == null
```

as if you really used the code

```
!ndt.HasValue
```

and similarly for code where you use not-equals and *null*. Basically, the C# compiler lets you treat the condition where *HasValue* is *null* as if the object itself were *null*. Even though it obviously isn't.

The C# compiler also fiddles with less-than and greater-than comparisons by generating code that calls *GetValueOrDefault* on both objects. (But if either but not both of the operands have *HasValue* properties of *false*, any less-than or greater-than comparison returns *false*.)

The C# compiler also simplifies the syntax for defining a nullable type. Rather than

```
Nullable<DateTime> ndt;
```

you can use the alias:

```
DateTime? ndt;
```

You don't need to use any of the constructors, properties, and methods provided by the *Nullable* class, and you can instead use simplified C# syntax instead. This statement defines a nullable *bool* and sets it to *null*:

```
bool? nb = null;
```

Or you can set it to a non-*null* value:

```
bool? nb = true;
```

Or you can set it from a non-nullable *bool*:

```
bool? nb = IsEnabled;
```

You can test the object against *null*, and you can use it in expressions with casting:

```
if (nb != null)
{
    if ((bool)nb)
    {
        ... // true case
    }
    else
    {
        ... // false case
    }
}
```

Notice that you can put a nullable *bool* in an *if* statement directly:

```
if (nb)    // Won't work!
```

A new operator has been added to C# to help work with nullable types. This is the *null coalescing operator*, which consists of a pair of question marks. Suppose *ndt* is a nullable *DateTime* object. You can assign an object of type DateTime like this:

```
DateTime dt = ndt ?? new DateTime(2007, 1, 1);
```

If *ndt* is not *null*, then *dt* is set to *ndt.Value*. Otherwise, *ndt* is set to a *DateTime* object of January 1, 2007.

Besides the *Nullable* generic structure, the *System* class also contains a *Nullable* class. This is a static class that you can use to compare two objects based on nullable types, and it also has a static method named *GetUnderlyingType* that you can use in connection with reflection.

At first it was believed that nullable types could be implemented without any changes to the Common Language Runtime. But that proved not to be the case. Suppose you have a nullable *int*:

```
int? ni;
```

And then there's code that might set *ni* to a value or might set it to *null*, and then *ni* is cast to an object of type *object*:

```
object obj = ni;
```

Or perhaps this casting happens when *ni* is passed to a method

```
bool IsNull = TestForNull(ni);
```

and the method is defined with an argument of type *object*:

```
bool TestForNull(object obj)
{
    return obj == null;
}
```

In either case, a boxing operation occurs, and the way the CLR works, boxed value types are never *null*. So even if *ni* is set to *null*, when it's cast to *obj*, *obj* is non-*null*.

As you might imagine, this behavior was considered undesirable, so the CLR was changed. Now when an instance of a nullable type is cast to an object of type *object*, and the *HasValue* property is *false*, the object of type *object* will be *null*.