2

Core C#

WHAT'S IN THIS CHAPTER?

- ➤ Declaring variables
- ➤ Initialization and scope of variables
- ➤ Predefined C# data types
- ➤ Dictating the flow of execution within a C# program using conditional statements, loops, and jump statements
- ➤ Enumerations
- ➤ Namespaces
- ➤ The `Main()` method
- ➤ Basic command-line C# compiler options
- ➤ Using `System.Console` to perform console I/O
- ➤ Using internal comments and documentation features
- ➤ Preprocessor directives
- ➤ Guidelines and conventions for good programming in C#

Now that you understand more about what C# can do, you will want to learn how to use it. This chapter gives you a good start in that direction by providing you with a basic knowledge of the fundamentals of C# programming, which is built on in subsequent chapters. By the end of this chapter, you will know enough C# to write simple programs (though without using inheritance or other object-oriented features, which are covered in later chapters).

YOUR FIRST C# PROGRAM

Let's start by compiling and running the simplest possible C# program — a simple console app consisting of a class that writes a message to the screen.

Later chapters present a number of code samples. The most common technique for writing C# programs is to use Visual Studio 2010 to generate a basic project and add your own code to it. However, because the aim of Part I is to teach the C# language, we are going to keep things simple and avoid relying on Visual Studio 2010 until Chapter 16, "Visual Studio 2010." Instead, we will present the code as simple files that you can type in using any text editor and compile from the command line.

The Code

Type the following into a text editor (such as Notepad), and save it with a `.cs` extension (for example, `First.cs`). The `Main()` method is shown here (for more information, see "The Main Method" section later in this chapter):

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   public class MyFirstClass
   {
      static void Main()
      {
         Console.WriteLine("Hello from Wrox.");
         Console.ReadLine();
         return;
      }
   }
}
```

code snippet First.cs

Compiling and Running the Program

You can compile this program by simply running the C# command-line compiler (`csc.exe`) against the source file, like this:

```
csc First.cs
```

If you want to compile code from the command line using the `csc` command, you should be aware that the .NET command-line tools, including `csc`, are available only if certain environment variables have been set up. Depending on how you installed .NET (and Visual Studio 2010), this may or may not be the case on your machine.

If you do not have the environment variables set up, you have the following two options: The first is to run the batch file `%Microsoft Visual Studio 2010%\Common7\Tools\vsvars32.bat` *from the command prompt before running* `csc`*, where* `%Microsoft Visual Studio 2010%` *is the folder to which Visual Studio 2010 has been installed. The second, and easier, way is to use the Visual Studio 2010 command prompt instead of the usual command prompt window. You will find the Visual Studio 2010 command prompt in the Start menu, under Programs, Microsoft Visual Studio 2010, Visual Studio Tools. It is simply a command prompt window that automatically runs* `vsvars32.bat` *when it opens.*

Compiling the code produces an executable file named `First.exe`, which you can run from the command line or from Windows Explorer like any other executable. Give it a try:

```
csc First.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.

First.exe
Hello from Wrox.
```

A Closer Look

First, a few general comments about C# syntax. In C#, as in other C-style languages, most statements end in a semicolon (`;`) and can continue over multiple lines without needing a continuation character. Statements can be joined into blocks using curly braces (`{}`). Single-line comments begin with two forward slash characters (`//`), and multiline comments begin with a slash and an asterisk (`/*`) and end with the same combination reversed (`*/`). In these aspects, C# is identical to C++ and Java but different from Visual Basic. It is the semicolons and curly braces that give C# code such a different visual appearance from Visual Basic code. If your background is predominantly Visual Basic, take extra care to remember the semicolon at the end of every statement. Omitting this is usually the biggest single cause of compilation errors among developers new to C-style languages. Another thing to remember is that C# is case-sensitive. That means that variables named `myVar` and `MyVar` are two different variables.

The first few lines in the previous code example have to do with *namespaces* (mentioned later in this chapter), which is a way to group together associated classes. The `namespace` keyword declares the namespace your class should be associated with. All code within the braces that follow it is regarded as being within that namespace. The `using` statement specifies a namespace that the compiler should look at to find any classes that are referenced in your code but that aren't defined in the current namespace. This serves the same purpose as the `import` statement in Java and the `using namespace` statement in C++.

```
using System;

namespace Wrox
{
```

The reason for the presence of the `using` statement in the `First.cs` file is that you are going to use a library class, `System.Console`. The `using System` statement allows you to refer to this class simply as `Console` (and similarly for any other classes in the `System` namespace). Without the `using`, we would have to fully qualify the call to the `Console.WriteLine` method like this:

```
System.Console.WriteLine("Hello from Wrox.");
```

The standard `System` namespace is where the most commonly used .NET types reside. It is important to realize that everything you do in C# depends on the .NET base classes. In this case, you are using the `Console` class within the `System` namespace to write to the console window. C# has no built-in keywords of its own for input or output; it is completely reliant on the .NET classes.

Because almost every C# program uses classes in the `System` *namespace, we will assume that a* `using System;` *statement is present in the file for all code snippets in this chapter.*

Next, you declare a class called `MyFirstClass`. However, because it has been placed in a namespace called `Wrox`, the fully qualified name of this class is `Wrox.MyFirstCSharpClass`:

```
    class MyFirstCSharpClass
    {
```

All C# code must be contained within a class. The class declaration consists of the `class` keyword, followed by the class name and a pair of curly braces. All code associated with the class should be placed between these braces.

Next, you declare a method called `Main()`. Every C# executable (such as console applications, Windows applications, and Windows services) must have an entry point — the `Main()` method (note the capital `M`):

```
        public static void Main()
        {
```

The method is called when the program is started. This method must return either nothing (`void`) or an integer (`int`). Note the format of method definitions in C#:

```
     [modifiers] return_type MethodName([parameters])
    {
       // Method body. NB. This code block is pseudo-code.
    }
```

Here, the first square brackets represent certain optional keywords. Modifiers are used to specify certain features of the method you are defining, such as where the method can be called from. In this case, you have two modifiers: `public` and `static`. The `public` modifier means that the method can be accessed from anywhere, so it can be called from outside your class. The `static` modifier indicates that the method does not operate on a specific instance of your class and therefore is called without first instantiating the class. This is important because you are creating an executable rather than a class library. You set the return type to `void`, and in the example, you don't include any parameters.

Finally, we come to the code statements themselves:

```
            Console.WriteLine("Hello from Wrox.");
            Console.ReadLine();
            return;
```

In this case, you simply call the `WriteLine()` method of the `System.Console` class to write a line of text to the console window. `WriteLine()` is a `static` method, so you don't need to instantiate a `Console` object before calling it.

`Console.ReadLine()` reads user input. Adding this line forces the application to wait for the carriage return key to be pressed before the application exits, and, in the case of Visual Studio 2010, the console window disappears.

You then call `return` to exit from the method (also, because this is the `Main()` method, you exit the program as well). You specified `void` in your method header, so you don't return any values.

Now that you have had a taste of basic C# syntax, you are ready for more detail. Because it is virtually impossible to write any nontrivial program without *variables,* we will start by looking at variables in C#.

VARIABLES

You declare variables in C# using the following syntax:

```
datatype identifier;
```

For example:

```
int i;
```

This statement declares an `int` named `i`. The compiler won't actually let you use this variable in an expression until you have initialized it with a value.

After it has been declared, you can assign a value to the variable using the assignment operator, `=`:

```
i = 10;
```

You can also declare the variable and initialize its value at the same time:

```
int i = 10;
```

If you declare and initialize more than one variable in a single statement, all of the variables will be of the same data type:

```
int x = 10, y =20;   // x and y are both ints
```

To declare variables of different types, you need to use separate statements. You cannot assign different data types within a multiple variable declaration:

```
int x = 10;
bool y = true;               // Creates a variable that stores true or false
int x = 10, bool y = true;   // This won't compile!
```

Notice the `//` and the text after it in the preceding examples. These are comments. The `//` character sequence tells the compiler to ignore the text that follows on this line because it is for a human to better understand the program and not part of the program itself. We further explain comments in code later in this chapter.

Initialization of Variables

Variable initialization demonstrates an example of C#'s emphasis on safety. Briefly, the C# compiler requires that any variable be initialized with some starting value before you refer to that variable in an operation. Most modern compilers will flag violations of this as a warning, but the ever-vigilant C# compiler treats such violations as errors. This prevents you from unintentionally retrieving junk values from memory that is left over from other programs.

C# has two methods for ensuring that variables are initialized before use:

➤ Variables that are fields in a class or struct, if not initialized explicitly, are by default zeroed out when they are created (classes and structs are discussed later).

➤ Variables that are local to a method must be explicitly initialized in your code prior to any statements in which their values are used. In this case, the initialization doesn't have to happen when the variable is declared, but the compiler will check all possible paths through the method and will flag an error if it detects any possibility of the value of a local variable being used before it is initialized.

For example, you can't do the following in C#:

```
public static int Main()
{
   int d;
   Console.WriteLine(d);   // Can't do this! Need to initialize d before use
   return 0;
}
```

Notice that this code snippet demonstrates defining `Main()` so that it returns an `int` instead of `void`.

When you attempt to compile these lines, you will receive this error message:

```
Use of unassigned local variable 'd'
```

Consider the following statement:

```
Something objSomething;
```

In C#, this line of code would create only a *reference* for a `Something` object, but this reference would not yet actually refer to any object. Any attempt to call a method or property against this variable would result in an error.

Instantiating a reference object in C# requires use of the `new` keyword. You create a reference as shown in the previous example and then point the reference at an object allocated on the heap using the `new` keyword:

```
objSomething = new Something();   // This creates a Something on the heap
```

Type Inference

Type inference makes use of the `var` keyword. The syntax for declaring the variable changes somewhat. The compiler "infers" what type the variable is by what the variable is initialized to. For example,

```
int someNumber = 0;
```

becomes

```
var someNumber = 0;
```

Even though `someNumber` is never declared as being an `int`, the compiler figures this out and `someNumber` is an `int` for as long as it is in scope. Once compiled, the two preceding statements are equal.

Here is a short program to demonstrate:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
  class Program
  {
    static void Main(string[] args)
    {
      var name = "Bugs Bunny";
      var age = 25;
      var isRabbit = true;

      Type nameType = name.GetType();
      Type ageType = age.GetType();
      Type isRabbitType = isRabbit.GetType();

      Console.WriteLine("name is type " + nameType.ToString());
      Console.WriteLine("age is type " + ageType.ToString());
      Console.WriteLine("isRabbit is type " + isRabbitType.ToString());
    }
  }
}
```

code snippet Var.cs

The output from this program is:

```
name is type System.String
age is type System.Int32
isRabbit is type System.Bool
```

There are a few rules that you need to follow:

- ➤ The variable must be initialized. Otherwise, the compiler doesn't have anything to infer the type from.
- ➤ The initializer cannot be null.
- ➤ The initializer must be an expression.
- ➤ You can't set the initializer to an object unless you create a new object in the initializer.

We examine this more closely in the discussion of anonymous types in Chapter 3, "Objects and Types."

After the variable has been declared and the type inferred, the variable's type cannot be changed. When established, the variable's type follows all the strong typing rules that any other variable type must follow.

Variable Scope

The *scope* of a variable is the region of code from which the variable can be accessed. In general, the scope is determined by the following rules:

- A *field* (also known as a member variable) of a class is in scope for as long as its containing class is in scope.
- A *local variable* is in scope until a closing brace indicates the end of the block statement or method in which it was declared.
- A local variable that is declared in a `for`, `while`, or similar statement is in scope in the body of that loop.

Scope Clashes for Local Variables

It's common in a large program to use the same variable name for different variables in different parts of the program. This is fine as long as the variables are scoped to completely different parts of the program so that there is no possibility for ambiguity. However, bear in mind that local variables with the same name can't be declared twice in the same scope. For example, you can't do this:

```
int x = 20;
// some more code
int x = 30;
```

Consider the following code sample:

Available for download on Wrox.com

```
using System;

namespace Wrox.ProCSharp.Basics
{
   public class ScopeTest
   {
      public static int Main()
      {
         for (int i = 0; i < 10; i++)
         {
            Console.WriteLine(i);
         }   // i goes out of scope here

         // We can declare a variable named i again, because
         // there's no other variable with that name in scope


         for (int i = 9; i >= 0; i--)
         {
            Console.WriteLine(i);
         }   // i goes out of scope here.
         return 0;
      }
   }
}
```

code snippet Scope.cs

This code simply prints out the numbers from 0 to 9, and then back again from 9 to 0, using two `for` loops. The important thing to note is that you declare the variable `i` twice in this code, within the same method. You can do this because `i` is declared in two separate loops, so each `i` variable is local to its own loop.

Here's another example:

Available for download on Wrox.com

```
public static int Main()
{
   int j = 20;
   for (int i = 0; i < 10; i++)
   {
      int j = 30;   // Can't do this—j is still in scope
      Console.WriteLine(j + i);
   }
   return 0;
}
```

code snippet ScopeBad.cs

If you try to compile this, you'll get an error like the following:

```
ScopeTest.cs(12,15): error CS0136: A local variable named 'j' cannot be declared in
this scope because it would give a different meaning to 'j', which is already used
in a 'parent or current' scope to denote something else.
```

This occurs because the variable `j`, which is defined before the start of the `for` loop, is still in scope within the `for` loop, and won't go out of scope until the `Main()` method has finished executing. Although the second `j` (the illegal one) is in the loop's scope, that scope is nested within the `Main()` method's scope. The compiler has no way to distinguish between these two variables, so it won't allow the second one to be declared.

Scope Clashes for Fields and Local Variables

In certain circumstances, however, you can distinguish between two identifiers with the same name (although not the same fully qualified name) and the same scope, and in this case the compiler allows you to declare the second variable. The reason is that C# makes a fundamental distinction between variables that are declared at the type level (fields) and variables that are declared within methods (local variables).

Consider the following code snippet:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class ScopeTest2
   {
       static int j = 20;
       public static void Main()
       {
           int j = 30;
           Console.WriteLine(j);
           return;
       }
    }
}
```

code snippet ScopeTest2.cs

This code will compile, even though you have two variables named `j` in scope within the `Main()` method: the `j` that was defined at the class level, and doesn't go out of scope until the class is destroyed (when the `Main()` method terminates, and the program ends); and the `j` defined in `Main()`. In this case, the new variable named `j` that you declare in the `Main()` method *hides* the class-level variable with the same name, so when you run this code, the number `30` will be displayed.

However, what if you want to refer to the class-level variable? You can actually refer to fields of a class or struct from outside the object, using the syntax `object.fieldname`. In the previous example, you are accessing a static field (you look at what this means in the next section) from a static method, so you can't use an instance of the class; you just use the name of the class itself:

```
...
public static void Main()
{
   int j = 30;
   Console.WriteLine(j);
   Console.WriteLine(ScopeTest2.j);
}
...
```

If you were accessing an instance field (a field that belongs to a specific instance of the class), you would need to use the `this` keyword instead.

Constants

As the name implies, a constant is a variable whose value cannot be changed throughout its lifetime. Prefixing a variable with the `const` keyword when it is declared and initialized designates that variable as a constant:

```
const int a = 100;   // This value cannot be changed.
```

Constants have the following characteristics:

- They must be initialized when they are declared, and after a value has been assigned, it can never be overwritten.
- The value of a constant must be computable at compile time. Therefore, you can't initialize a constant with a value taken from a variable. If you need to do this, you will need to use a read-only field (this is explained in Chapter 3).
- Constants are always implicitly static. However, notice that you don't have to (and, in fact, are not permitted to) include the `static` modifier in the constant declaration.

At least three advantages exist for using constants in your programs:

- Constants make your programs easier to read by replacing magic numbers and strings with readable names whose values are easy to understand.
- Constants make your programs easier to modify. For example, assume that you have a `SalesTax` constant in one of your C# programs, and that constant is assigned a value of 6 percent. If the sales tax rate changes at a later point in time, you can modify the behavior of all tax calculations simply by assigning a new value to the constant; you don't have to hunt through your code for the value `.06` and change each one, hoping that you will find all of them.
- Constants help prevent mistakes in your programs. If you attempt to assign another value to a constant somewhere in your program other than at the point where the constant is declared, the compiler will flag the error.

PREDEFINED DATA TYPES

Now that you have seen how to declare variables and constants, let's take a closer look at the data types available in C#. As you will see, C# is much stricter about the types available and their definitions than some other languages are.

Value Types and Reference Types

Before examining the data types in C#, it is important to understand that C# distinguishes between two categories of data type:

- Value types
- Reference types

The next few sections look in detail at the syntax for value and reference types. Conceptually, the difference is that a *value type* stores its value directly, whereas a *reference type* stores a reference to the value.

These types are stored in different places in memory; value types are stored in an area known as the *stack*, and reference types are stored in an area known as the *managed heap*. It is important to be aware of whether a type is a value type or a reference type because of the different effect each assignment has. For example, `int` is a value type, which means that the following statement will result in two locations in memory storing the value 20:

```
// i and j are both of type int
i = 20;
j = i;
```

However, consider the following code. For this code, assume that you have defined a class called `Vector`. Assume that `Vector` is a reference type and has an `int` member variable called `Value`:

```
Vector x, y;
x = new Vector();
x.Value = 30;   // Value is a field defined in Vector class
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

The crucial point to understand is that after executing this code, there is only one `Vector` object around. `x` and `y` both point to the memory location that contains this object. Because `x` and `y` are variables of a reference type, declaring each variable simply reserves a reference — it doesn't instantiate an object of the given type. In neither case is an object actually created. To create an object, you have to use the `new` keyword, as shown. Because `x` and `y` refer to the same object, changes made to `x` will affect `y` and vice versa. Hence the code will display `30` and then `50`.

C++ developers should note that this syntax is like a reference, not a pointer. You use the . notation, not ->, to access object members. Syntactically, C# references look more like C++ reference variables. However, behind the superficial syntax, the real similarity is with C++ pointers.

If a variable is a reference, it is possible to indicate that it does not refer to any object by setting its value to `null`:

```
y = null;
```

If a reference is set to `null`, then clearly it is not possible to call any nonstatic member functions or fields against it; doing so would cause an exception to be thrown at runtime.

In C#, basic data types such as `bool` and `long` are value types. This means that if you declare a `bool` variable and assign it the value of another `bool` variable, you will have two separate `bool` values in memory. Later, if you change the value of the original `bool` variable, the value of the second `bool` variable does not change. These types are copied by value.

In contrast, most of the more complex C# data types, including classes that you yourself declare, are reference types. They are allocated upon the heap, have lifetimes that can span multiple function calls, and can be accessed through one or several aliases. The Common Language Runtime (CLR) implements an elaborate algorithm to track which reference variables are still reachable and which have been orphaned. Periodically, the CLR will destroy orphaned objects and return the memory that they once occupied back to the operating system. This is done by the garbage collector.

C# has been designed this way because high performance is best served by keeping primitive types (such as `int` and `bool`) as value types and larger types that contain many fields (as is usually the case with classes) as reference types. If you want to define your own type as a value type, you should declare it as a struct.

CTS Types

As mentioned in Chapter 1, ".NET Architecture," the basic predefined types recognized by C# are not intrinsic to the language but are part of the .NET Framework. For example, when you declare an `int` in C#, what you are actually declaring is an instance of a .NET struct, `System.Int32`. This may sound like a small point, but it has a profound significance: it means that you are able to treat all the primitive data types syntactically as if they were classes that supported certain methods. For example, to convert an `int i` to a `string`, you can write:

```
string s = i.ToString();
```

It should be emphasized that, behind this syntactical convenience, the types really are stored as primitive types, so there is absolutely no performance cost associated with the idea that the primitive types are notionally represented by .NET structs.

The following sections review the types that are recognized as built-in types in C#. Each type is listed, along with its definition and the name of the corresponding .NET type (CTS type). C# has 15 predefined types, 13 value types, and 2 (`string` and `object`) reference types.

Predefined Value Types

The built-in CTS value types represent primitives, such as integer and floating-point numbers, character, and Boolean types.

Integer Types

C# supports eight predefined integer types, shown in the following table.

NAME	CTS TYPE	DESCRIPTION	RANGE (MIN:MAX)
`sbyte`	`System.SByte`	8-bit signed integer	-128:127 ($-2^7:2^7-1$)
`short`	`System.Int16`	16-bit signed integer	-32,768:32,767 ($-2^{15}:2^{15}-1$)
`int`	`System.Int32`	32-bit signed integer	-2,147,483,648:2,147,483,647 ($-2^{31}:2^{31}-1$)
`long`	`System.Int64`	64-bit signed integer	-9,223,372,036,854,775,808: 9,223,372,036,854 ,775,807 ($-2^{63}:2^{63}-1$)
`byte`	`System.Byte`	8-bit unsigned integer	0:255 ($0:2^8-1$)
`ushort`	`System.UInt16`	16-bit unsigned integer	0:65,535 ($0:2^{16}-1$)
`uint`	`System.UInt32`	32-bit unsigned integer	0:4,294,967,295 ($0:2^{32}-1$)
`ulong`	`System.UInt64`	64-bit unsigned integer	0:18,446,744,073,709,551,615 ($0:2^{64}-1$)

Some C# types have the same names as C++ and Java types but have different definitions. For example, in C#, an `int` is always a 32-bit signed integer. In C++ an `int` is a signed integer, but the number of bits is platform-dependent (32 bits on Windows). In C#, all data types have been defined in a platform-independent manner to allow for the possible future porting of C# and .NET to other platforms.

A `byte` is the standard 8-bit type for values in the range 0 to 255 inclusive. Be aware that, in keeping with its emphasis on type safety, C# regards the `byte` type and the `char` type as completely distinct, and any programmatic conversions between the two must be explicitly requested. Also be aware that unlike the other types in the integer family, a `byte` type is by default unsigned. Its signed version bears the special name `sbyte`.

With .NET, a `short` is no longer quite so short; it is now 16 bits long. The `int` type is 32 bits long. The `long` type reserves 64 bits for values. All integer-type variables can be assigned values in decimal or in hex notation. The latter requires the `0x` prefix:

```
long x = 0x12ab;
```

If there is any ambiguity about whether an integer is `int`, `uint`, `long`, or `ulong`, it will default to an `int`. To specify which of the other integer types the value should take, you can append one of the following characters to the number:

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

You can also use lowercase `u` and `l`, although the latter could be confused with the integer `1` (one).

Floating-Point Types

Although C# provides a plethora of integer data types, it supports floating-point types as well.

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
`float`	`System.Single`	32-bit single-precision floating point	7	±1.5 × 10^{245} to ±3.4 × 10^{38}
`double`	`System.Double`	64-bit double-precision floating point	15/16	±5.0 × 10^{2324} to ±1.7 × 10^{308}

The `float` data type is for smaller floating-point values, for which less precision is required. The `double` data type is bulkier than the `float` data type but offers twice the precision (15 digits).

If you hard-code a non-integer number (such as 12.3) in your code, the compiler will normally assume that you want the number interpreted as a `double`. If you want to specify that the value is a `float`, you append the character `F` (or `f`) to it:

```
float f = 12.3F;
```

The Decimal Type

The `decimal` type represents higher-precision floating-point numbers, as shown in the following table.

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
`decimal`	`System.Decimal`	128-bit high-precision decimal notation	28	±1.0 × 10^{228} to ±7.9 × 10^{28}

One of the great things about the CTS and C# is the provision of a dedicated `decimal` type for financial calculations. How you use the 28 digits that the decimal type provides is up to you. In other words, you can track smaller dollar amounts with greater accuracy for cents or larger dollar amounts with more rounding in the fractional area. Bear in mind, however, that `decimal` is not implemented under the hood as a primitive type, so using `decimal` will have a performance effect on your calculations.

To specify that your number is a `decimal` type rather than a `double`, `float`, or an integer, you can append the `M` (or `m`) character to the value, as shown in the following example:

```
decimal d = 12.30M;
```

The Boolean Type

The C# bool type is used to contain Boolean values of either true or false.

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
bool	System.Boolean	Represents true or false	NA	true false

You cannot implicitly convert bool values to and from integer values. If a variable (or a function return type) is declared as a bool, you can only use values of true and false. You will get an error if you try to use zero for false and a non-zero value for true.

The Character Type

For storing the value of a single character, C# supports the char data type.

NAME	CTS TYPE	VALUES
char	System.Char	Represents a single 16-bit (Unicode) character

Literals of type char are signified by being enclosed in single quotation marks, for example 'A'. If you try to enclose a character in double quotation marks, the compiler will treat this as a string and throw an error.

As well as representing chars as character literals, you can represent them with four-digit hex Unicode values (for example '\u0041'), as integer values with a cast (for example, (char)65), or as hexadecimal values ('\x0041'). You can also represent them with an escape sequence, as shown in the following table.

ESCAPE SEQUENCE	CHARACTER
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab character
\v	Vertical tab

Predefined Reference Types

C# supports two predefined reference types, object and string, described in the following table.

NAME	CTS TYPE	DESCRIPTION
object	System.Object	The root type. All other types in the CTS are derived (including value types) from object.
string	System.String	Unicode character string

The object Type

Many programming languages and class hierarchies provide a root type, from which all other objects in the hierarchy are derived. C# and .NET are no exception. In C#, the `object` type is the ultimate parent type from which all other intrinsic and user-defined types are derived. This means that you can use the `object` type for two purposes:

- You can use an `object` reference to bind to an object of any particular subtype. For example, in Chapter 7, "Operators and Casts," you will see how you can use the `object` type to box a value object on the stack to move it to the heap. `object` references are also useful in reflection, when code must manipulate objects whose specific types are unknown.
- The `object` type implements a number of basic, general-purpose methods, which include `Equals()`, `GetHashCode()`, `GetType()`, and `ToString()`. Responsible user-defined classes may need to provide replacement implementations of some of these methods using an object-oriented technique known as *overriding,* which is discussed in Chapter 4, "Inheritance." When you override `ToString()`, for example, you equip your class with a method for intelligently providing a string representation of itself. If you don't provide your own implementations for these methods in your classes, the compiler will pick up the implementations in `object`, which may or may not be correct or sensible in the context of your classes.

We examine the `object` type in more detail in subsequent chapters.

The string Type

C# recognizes the `string` keyword, which under the hood is translated to the .NET class, `System.String`. With it, operations like string concatenation and string copying are a snap:

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

Despite this style of assignment, `string` is a reference type. Behind the scenes, a `string` object is allocated on the heap, not the stack, and when you assign one string variable to another string, you get two references to the same string in memory. However, with `string` there are some differences from the usual behavior for reference types. For example, strings are immutable. Should you make changes to one of these strings, this will create an entirely new `string` object, leaving the other string unchanged. Consider the following code:

Available for download on Wrox.com

```
using System;

class StringExample
{
   public static int Main()
   {
      string s1 = "a string";
      string s2 = s1;
      Console.WriteLine("s1 is " + s1);
      Console.WriteLine("s2 is " + s2);
      s1 = "another string";
      Console.WriteLine("s1 is now " + s1);
      Console.WriteLine("s2 is now " + s2);
      return 0;
   }
}
```

code snippet StringExample.cs

The output from this is:

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

Changing the value of `s1` has no effect on `s2`, contrary to what you'd expect with a reference type! What's happening here is that when `s1` is initialized with the value `a string`, a new string object is allocated on the heap. When `s2` is initialized, the reference points to this same object, so `s2` also has the value `a string`. However, when you now change the value of `s1`, instead of replacing the original value, a new object will be allocated on the heap for the new value. The `s2` variable will still point to the original object, so its value is unchanged. Under the hood, this happens as a result of operator overloading, a topic that is explored in Chapter 7. In general, the `string` class has been implemented so that its semantics follow what you would normally intuitively expect for a string.

String literals are enclosed in double quotation marks (`"."`); if you attempt to enclose a string in single quotation marks, the compiler will take the value as a `char`, and throw an error. C# strings can contain the same Unicode and hexadecimal escape sequences as `char`s. Because these escape sequences start with a backslash, you can't use this character unescaped in a string. Instead, you need to escape it with two backslashes (`\\`):

```
string filepath = "C:\\ProCSharp\\First.cs";
```

Even if you are confident that you can remember to do this all the time, typing all those double backslashes can prove annoying. Fortunately, C# gives you an alternative. You can prefix a string literal with the at character (`@`) and all the characters in it will be treated at face value; they won't be interpreted as escape sequences:

```
string filepath = @"C:\ProCSharp\First.cs";
```

This even allows you to include line breaks in your string literals:

```
string jabberwocky = @"'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.";
```

Then the value of `jabberwocky` would be this:

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

FLOW CONTROL

This section looks at the real nuts and bolts of the language: the statements that allow you to control the *flow* of your program rather than executing every line of code in the order it appears in the program.

Conditional Statements

Conditional statements allow you to branch your code depending on whether certain conditions are met or on the value of an expression. C# has two constructs for branching code — the `if` statement, which allows you to test whether a specific condition is met, and the `switch` statement, which allows you to compare an expression with a number of different values.

The if Statement

For conditional branching, C# inherits the C and C++ `if.else` construct. The syntax should be fairly intuitive for anyone who has done any programming with a procedural language:

```
if (condition)
   statement(s)
else
   statement(s)
```

If more than one statement is to be executed as part of either condition, these statements need to be joined together into a block using curly braces (`{.}`). (This also applies to other C# constructs where statements can be joined into a block, such as the `for` and `while` loops):

```
bool isZero;
if (i == 0)
{
   isZero = true;
   Console.WriteLine("i is Zero");
}
else
{
   isZero = false;
   Console.WriteLine("i is Non-zero");
}
```

If you want to, you can use an `if` statement without a final `else` statement. You can also combine `else if` clauses to test for multiple conditions:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class MainEntryPoint
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Type in a string");
         string input;
         input = Console.ReadLine();
         if (input == "")
         {
            Console.WriteLine("You typed in an empty string.");
         }
         else if (input.Length < 5)
         {
            Console.WriteLine("The string had less than 5 characters.");
         }
         else if (input.Length < 10)
         {
            Console.WriteLine("The string had at least 5 but less than 10
               Characters.");
         }
         Console.WriteLine("The string was " + input);
      }
   }
```

code snippet ElseIf.cs

There is no limit to how many `else if`s you can add to an `if` clause.

You'll notice that the previous example declares a string variable called `input`, gets the user to enter text at the command line, feeds this into `input`, and then tests the length of this string variable. The code also shows how easy string manipulation can be in C#. To find the length of `input`, for example, use `input.Length`.

One point to note about `if` is that you don't need to use the braces if there's only one statement in the conditional branch:

```
if (i == 0) Let's add some brackets here.
   Console.WriteLine("i is Zero");       // This will only execute if i == 0
Console.WriteLine("i can be anything");  // Will execute whatever the
                                         // value of i
```

However, for consistency, many programmers prefer to use curly braces whenever they use an `if` statement.

The `if` statements presented also illustrate some of the C# operators that compare values. Note in particular that C# uses == to compare variables for equality. Do not use = for this purpose. A single = is used to assign values.

In C#, the expression in the `if` clause must evaluate to a Boolean. It is not possible to test an integer (returned from a function, say) directly. You have to convert the integer that is returned to a Boolean `true` or `false`, for example, by comparing the value with zero or with `null`:

```
if (DoSomething() != 0)
{
   // Non-zero value returned
}
else
{
   // Returned zero
}
```

The switch Statement

The `switch.case` statement is good for selecting one branch of execution from a set of mutually exclusive ones. It takes the form of a `switch` argument followed by a series of `case` clauses. When the expression in the `switch` argument evaluates to one of the values beside a `case` clause, the code immediately following the `case` clause executes. This is one example where you don't need to use curly braces to join statements into blocks; instead, you mark the end of the code for each case using the `break` statement. You can also include a `default` case in the `switch` statement, which will execute if the expression evaluates to none of the other cases. The following `switch` statement tests the value of the `integerA` variable:

```
switch (integerA)
{
   case 1:
      Console.WriteLine("integerA =1");
      break;
   case 2:
      Console.WriteLine("integerA =2");
      break;
   case 3:
      Console.WriteLine("integerA =3");
      break;
   default:
      Console.WriteLine("integerA is not 1,2, or 3");
      break;
}
```

Note that the case values must be constant expressions; variables are not permitted.

Though the `switch.case` statement should be familiar to C and C++ programmers, C#'s `switch.case` is a bit safer than its C++ equivalent. Specifically, it prohibits fall-through conditions in almost all cases. This means that if a `case` clause is fired early on in the block, later clauses cannot be fired unless you use a `goto` statement to mark that you want them fired, too. The compiler enforces this restriction by flagging every `case` clause that is not equipped with a `break` statement as an error similar to this:

```
Control cannot fall through from one case label ('case 2:') to another
```

Although it is true that fall-through behavior is desirable in a limited number of situations, in the vast majority of cases, it is unintended and results in a logical error that's hard to spot. Isn't it better to code for the norm rather than for the exception?

By getting creative with `goto` statements, however, you can duplicate fall-through functionality in your `switch.cases`. But, if you find yourself really wanting to, you probably should reconsider your approach.

The following code illustrates both how to use `goto` to simulate fall-through, and how messy the resultant code can get:

```
// assume country and language are of type string
switch(country)
{
   case "America":
      CallAmericanOnlyMethod();
      goto case "Britain";
   case "France":
      language = "French";
      break;
   case "Britain":
      language = "English";
      break;
}
```

There is one exception to the no-fall-through rule, however, in that you can fall through from one case to the next if that case is empty. This allows you to treat two or more cases in an identical way (without the need for `goto` statements):

```
switch(country)
{
   case "au":
   case "uk":
   case "us":
      language = "English";
      break;
   case "at":
   case "de":
      language = "German";
      break;
}
```

One intriguing point about the `switch` statement in C# is that the order of the cases doesn't matter — you can even put the `default` case first! As a result, no two cases can be the same. This includes different constants that have the same value, so you can't, for example, do this:

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
   case england:
   case britain:    // This will cause a compilation error.
      language = "English";
      break;
}
```

The previous code also shows another way in which the `switch` statement is different in C# compared to C++: In C#, you are allowed to use a string as the variable being tested.

Loops

C# provides four different loops (`for`, `while`, do . . . while, and `foreach`) that allow you to execute a block of code repeatedly until a certain condition is met.

The for Loop

C# `for` loops provide a mechanism for iterating through a loop where you test whether a particular condition holds before you perform another iteration. The syntax is:

```
for (initializer; condition; iterator)
   statement(s)
```

where:

- The initializer is the expression evaluated before the first loop is executed (usually initializing a local variable as a loop counter).
- The condition is the expression checked before each new iteration of the loop (this must evaluate to `true` for another iteration to be performed).
- The iterator is an expression evaluated after each iteration (usually incrementing the loop counter).

The iterations end when the condition evaluates to `false`.

The `for` loop is a so-called pretest loop because the loop condition is evaluated before the loop statements are executed, and so the contents of the loop won't be executed at all if the loop condition is `false`.

The `for` loop is excellent for repeating a statement or a block of statements for a predetermined number of times. The following example is typical of the use of a `for` loop. The following code will write out all the integers from 0 to 99:

```
for (int i = 0; i < 100; i=i+1)   // This is equivalent to
                                           // For i = 0 To 99 in VB.
{
   Console.WriteLine(i);
}
```

Here, you declare an `int` called `i` and initialize it to zero. This will be used as the loop counter. You then immediately test whether it is less than 100. Because this condition evaluates to `true`, you execute the code in the loop, displaying the value 0. You then increment the counter by one, and walk through the process again. Looping ends when `i` reaches 100.

Actually, the way the preceding loop is written isn't quite how you would normally write it. C# has a shorthand for adding 1 to a variable, so instead of `i = i + 1`, you can simply write `i++`:

```
for (int i = 0; i < 100; i++)
{
    // etc.
       }
```

You can also make use of type inference for the iteration variable `i` in the preceding example. Using type inference the loop construct would be:

```
for (var i = 0; i < 100; i++)
...
```

It's not unusual to nest `for` loops so that an inner loop executes once completely for each iteration of an outer loop. This scheme is typically employed to loop through every element in a rectangular multidimensional array. The outermost loop loops through every row, and the inner loop loops through every column in a particular row. The following code displays rows of numbers. It also uses another `Console` method, `Console.Write()`, which does the same as `Console.WriteLine()` but doesn't send a carriage return to the output.

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class MainEntryPoint
   {
      static void Main(string[] args)
      {
         // This loop iterates through rows
         for (int i = 0; i < 100; i+=10)
         {
            // This loop iterates through columns
            for (int j = i; j < i + 10; j++)
            {
               Console.Write(" " + j);
            }
            Console.WriteLine();
         }
      }
   }
}
```

code snippet NestedFor.cs

Although `j` is an integer, it will be automatically converted to a string so that the concatenation can take place.

The preceding sample results in this output:

```
 0  1  2  3  4  5  6  7  8  9
10  11  12  13  14  15  16  17  18  19
20  21  22  23  24  25  26  27  28  29
30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49
50  51  52  53  54  55  56  57  58  59
60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79
80  81  82  83  84  85  86  87  88  89
90  91  92  93  94  95  96  97  98  99
```

Although it is technically possible to evaluate something other than a counter variable in a `for` loop's test condition, it is certainly not typical. It is also possible to omit one (or even all) of the expressions in the `for` loop. In such situations, however, you should consider using the `while` loop.

The while Loop

Like the `for` loop, `while` is a pretest loop. The syntax is similar, but `while` loops take only one expression:

```
while(condition)
   statement(s);
```

Unlike the `for` loop, the `while` loop is most often used to repeat a statement or a block of statements for a number of times that is not known before the loop begins. Usually, a statement inside the `while` loop's body will set a Boolean flag to `false` on a certain iteration, triggering the end of the loop, as in the following example:

```
bool condition = false;
while (!condition)
{
   // This loop spins until the condition is true.
   DoSomeWork();
   condition = CheckCondition();   // assume CheckCondition() returns a bool
}
```

The do . . . while Loop

The `do...while` loop is the post-test version of the `while` loop. This means that the loop's test condition is evaluated after the body of the loop has been executed. Consequently, `do...while` loops are useful for situations in which a block of statements must be executed at least one time, as in this example:

```
bool condition;
do
{
   // This loop will at least execute once, even if Condition is false.
   MustBeCalledAtLeastOnce();
   condition = CheckCondition();
} while (condition);
```

The foreach Loop

The `foreach` loop allows you to iterate through each item in a collection. For now, we won't worry about exactly what a collection is (it is explained fully in Chapter 10, "Collections"); we will just say that it is an object that represents a list of objects. Technically, to count as a collection, it must support an interface called `IEnumerable`. Examples of collections include C# arrays, the collection classes in the `System.Collection` namespaces, and user-defined collection classes. You can get an idea of the syntax of `foreach` from the following code, if you assume that `arrayOfInts` is (unsurprisingly) an array of `int`s:

```
foreach (int temp in arrayOfInts)
{
   Console.WriteLine(temp);
}
```

Here, `foreach` steps through the array one element at a time. With each element, it places the value of the element in the `int` variable called `temp` and then performs an iteration of the loop.

Here is another situation where type inference can be used. The `foreach` loop would become:

```
foreach (var temp in arrayOfInts)
...
```

`temp` would be inferred to `int` because that is what the collection item type is.

An important point to note with `foreach` is that you can't change the value of the item in the collection (`temp` in the preceding code), so code such as the following will not compile:

```
foreach (int temp in arrayOfInts)
{
   temp++;
   Console.WriteLine(temp);
}
```

If you need to iterate through the items in a collection and change their values, you will need to use a `for` loop instead.

Jump Statements

C# provides a number of statements that allow you to jump immediately to another line in the program. The first of these is, of course, the notorious `goto` statement.

The goto Statement

The `goto` statement allows you to jump directly to another specified line in the program, indicated by a *label* (this is just an identifier followed by a colon):

```
goto Label1;
   Console.WriteLine("This won't be executed");
Label1:
   Console.WriteLine("Continuing execution from here");
```

A couple of restrictions are involved with `goto`. You can't jump into a block of code such as a `for` loop, you can't jump out of a class, and you can't exit a `finally` block after `try.catch` blocks (Chapter 15, "Errors and Exceptions," looks at exception handling with `try.catch.finally`).

The reputation of the `goto` statement probably precedes it, and in most circumstances, its use is sternly frowned upon. In general, it certainly doesn't conform to good object-oriented programming practice.

The break Statement

You have already met the `break` statement briefly — when you used it to exit from a case in a `switch` statement. In fact, `break` can also be used to exit from `for`, `foreach`, `while`, or `do...while` loops. Control will switch to the statement immediately after the end of the loop.

If the statement occurs in a nested loop, control will switch to the end of the innermost loop. If the break occurs outside of a `switch` statement or a loop, a compile-time error will occur.

The continue Statement

The `continue` statement is similar to `break`, and must also be used within a `for`, `foreach`, `while`, or `do...while` loop. However, it exits only from the current iteration of the loop, meaning that execution will restart at the beginning of the next iteration of the loop, rather than outside the loop altogether.

The return Statement

The `return` statement is used to exit a method of a class, returning control to the caller of the method. If the method has a return type, `return` must return a value of this type; otherwise if the method returns `void`, you should use `return` without an expression.

ENUMERATIONS

An *enumeration* is a user-defined integer type. When you declare an enumeration, you specify a set of acceptable values that instances of that enumeration can contain. Not only that, but you can give the values user-friendly names. If, somewhere in your code, you attempt to assign a value that is not in the acceptable set of values to an instance of that enumeration, the compiler will flag an error.

Creating an enumeration can save you a lot of time and headaches in the long run. At least three benefits exist to using enumerations instead of plain integers:

- As mentioned, enumerations make your code easier to maintain by helping to ensure that your variables are assigned only legitimate, anticipated values.
- Enumerations make your code clearer by allowing you to refer to integer values by descriptive names rather than by obscure “magic” numbers.
- Enumerations make your code easier to type, too. When you go to assign a value to an instance of an enumerated type, the Visual Studio .NET IDE will, through IntelliSense, pop up a list box of acceptable values to save you some keystrokes and to remind you of what the possible options are.

You can define an enumeration as follows:

```
public enum TimeOfDay
{
   Morning = 0,
   Afternoon = 1,
   Evening = 2
}
```

In this case, you use an integer value to represent each period of the day in the enumeration. You can now access these values as members of the enumeration. For example, `TimeOfDay.Morning` will return the value `0`. You will typically use this enumeration to pass an appropriate value into a method and iterate through the possible values in a `switch` statement:

```
class EnumExample
{
   public static int Main()
   {
      WriteGreeting(TimeOfDay.Morning);
      return 0;
   }

   static void WriteGreeting(TimeOfDay timeOfDay)
   {
      switch(timeOfDay)
```

```
         {
            case TimeOfDay.Morning:
               Console.WriteLine("Good morning!");
               break;
            case TimeOfDay.Afternoon:
               Console.WriteLine("Good afternoon!");
               break;
            case TimeOfDay.Evening:
               Console.WriteLine("Good evening!");
               break;
            default:
               Console.WriteLine("Hello!");
               break;
         }
      }
   }
```

The real power of enums in C# is that behind the scenes they are instantiated as structs derived from the base class, `System.Enum`. This means it is possible to call methods against them to perform some useful tasks. Note that because of the way the .NET Framework is implemented there is no performance loss associated with treating the enums syntactically as structs. In practice, after your code is compiled, enums will exist as primitive types, just like `int` and `float`.

You can retrieve the string representation of an enum as in the following example, using the earlier `TimeOfDay` enum:

```
TimeOfDay time = TimeOfDay.Afternoon;
Console.WriteLine(time.ToString());
```

This will return the string `Afternoon`.

Alternatively, you can obtain an enum value from a string:

```
TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);
```

This code snippet illustrates both obtaining an enum value from a string and converting to an integer. To convert from a string, you need to use the static `Enum.Parse()` method, which, as shown, takes three parameters. The first is the type of enum you want to consider. The syntax is the keyword `typeof` followed by the name of the enum class in brackets. (Chapter 7 explores the `typeof` operator in more detail.) The second parameter is the string to be converted, and the third parameter is a `bool` indicating whether you should ignore case when doing the conversion. Finally, note that `Enum.Parse()` actually returns an object reference — you need to explicitly convert this to the required enum type (this is an example of an unboxing operation). For the preceding code, this returns the value `1` as an object, corresponding to the enum value of `TimeOfDay.Afternoon`. On converting explicitly to an `int`, this produces the value `1` again.

Other methods on `System.Enum` do things such as return the number of values in an enum definition or list the names of the values. Full details are in the MSDN documentation.

NAMESPACES

As you saw earlier in this chapter, namespaces provide a way of organizing related classes and other types. Unlike a file or a component, a namespace is a logical rather than physical grouping. When you define a class in a C# file, you can include it within a namespace definition. Later, when you define another class that performs related work in another file, you can include it within the same namespace, creating a logical grouping that gives an indication to other developers using the classes how they are related and used:

```
namespace CustomerPhoneBookApp
{
   using System;

   public struct Subscriber
```

```
      {
         // Code for struct here...
      }
   }
```

Placing a type in a namespace effectively gives that type a long name, consisting of the type's namespace as a series of names separated with periods (`.`), terminating with the name of the class. In the preceding example, the full name of the `Subscriber` struct is `CustomerPhoneBookApp.Subscriber`. This allows distinct classes with the same short name to be used within the same program without ambiguity. This full name is often called the fully qualified name.

You can also nest namespaces within other namespaces, creating a hierarchical structure for your types:

```
namespace Wrox
{
   namespace ProCSharp
   {
      namespace Basics
      {
         class NamespaceExample
         {
            // Code for the class here...
         }
      }
   }
}
```

Each namespace name is composed of the names of the namespaces it resides within, separated with periods, starting with the outermost namespace and ending with its own short name. So the full name for the `ProCSharp` namespace is `Wrox.ProCSharp`, and the full name of `NamespaceExample` class is `Wrox .ProCSharp.Basics.NamespaceExample`.

You can use this syntax to organize the namespaces in your namespace definitions too, so the previous code could also be written as follows:

```
namespace Wrox.ProCSharp.Basics
{
   class NamespaceExample
   {
      // Code for the class here...
   }
}
```

Note that you are not permitted to declare a multipart namespace nested within another namespace.

Namespaces are not related to assemblies. It is perfectly acceptable to have different namespaces in the same assembly or to define types in the same namespace in different assemblies.

Defining the namespace hierarchy should be planned out prior to the start of a project. Generally the accepted format is `CompanyName.ProjectName.SystemSection`. So in the previous example, Wrox is the company name, ProCSharp is the project, and in the case of this chapter, Basics is the section.

The using Directive

Obviously, namespaces can grow rather long and tiresome to type, and the ability to indicate a particular class with such specificity may not always be necessary. Fortunately, as noted at the beginning of this chapter, C# allows you to abbreviate a class's full name. To do this, list the class's namespace at the top of the file, prefixed with the `using` keyword. Throughout the rest of the file, you can refer to the types in the namespace simply by their type names:

```
using System;
using Wrox.ProCSharp;
```

As remarked earlier, virtually all C# source code will have the statement `using System;` simply because so many useful classes supplied by Microsoft are contained in the `System` namespace.

If two namespaces referenced by `using` statements contain a type of the same name, you will need to use the full (or at least a longer) form of the name to ensure that the compiler knows which type is to be accessed. For example, say classes called `NamespaceExample` exist in both the `Wrox.ProCSharp.Basics` and `Wrox.ProCSharp.OOP` namespaces. If you then create a class called `Test` in the `Wrox.ProCSharp` namespace, and instantiate one of the `NamespaceExample` classes in this class, you need to specify which of these two classes you're talking about:

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;
namespace Wrox.ProCSharp
{
  class Test
  {
    public static int Main()
    {
      Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
     // do something with the nSEx variable.
      return 0;
    }
  }
|
```

Because `using` *statements occur at the top of C# files, in the same place that C and C++ list* `#include` *statements, it's easy for programmers moving from C++ to C# to confuse namespaces with C++-style header files. Don't make this mistake. The* `using` *statement does no physical linking between files, and C# has no equivalent to C++ header files.*

Your organization will probably want to spend some time developing a namespace schema so that its developers can quickly locate functionality that they need and so that the names of the organization's homegrown classes won't conflict with those in off-the-shelf class libraries. Guidelines on establishing your own namespace scheme along with other naming recommendations are discussed later in this chapter.

Namespace Aliases

Another use of the `using` keyword is to assign aliases to classes and namespaces. If you have a very long namespace name that you want to refer to several times in your code but don't want to include in a simple `using` statement (for example, to avoid type name conflicts), you can assign an alias to the namespace. The syntax for this is as follows:

```
using alias = NamespaceName;
```

The following example (a modified version of the previous example) assigns the alias `Introduction` to the `Wrox.ProCSharp.Basics` namespace and uses this to instantiate a `NamespaceExample` object, which is defined in this namespace. Notice the use of the namespace alias qualifier (`::`). This forces the search to start with the `Introduction` namespace alias. If a class called `Introduction` had been introduced in the same scope, a conflict would happen. The :: operator allows the alias to be referenced even if the conflict exists. The `NamespaceExample` class has one method, `GetNamespace()`, which uses the `GetType()` method exposed by every class to access a `Type` object representing the class's type. You use this object to return a name of the class's namespace:

```
using System;
using Introduction =  Wrox.ProCSharp.Basics;
```

```
class Test
{
   public static int Main()
   {
      Introduction::NamespaceExample NSEx =
         new Introduction::NamespaceExample();
      Console.WriteLine(NSEx.GetNamespace());
      return 0;
   }
}

namespace Wrox.ProCSharp.Basics
{
   class NamespaceExample
   {
      public string GetNamespace()
      {
         return this.GetType().Namespace;
      }
   }
}
```

THE MAIN() METHOD

As you saw at the start of this chapter, C# programs start execution at a method named `Main()`. This must be a static method of a class (or struct), and must have a return type of either `int` or `void`.

Although it is common to specify the `public` modifier explicitly, because by definition the method must be called from outside the program, it doesn't actually matter what accessibility level you assign to the entry-point method — it will run even if you mark the method as `private`.

Multiple Main() Methods

When a C# console or Windows application is compiled, by default the compiler looks for exactly one `Main()` method in any class matching the signature that was just described and makes that class method the entry point for the program. If there is more than one `Main()` method, the compiler will return an error message. For example, consider the following code called `DoubleMain.cs`:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class Client
   {
      public static int Main()
      {
         MathExample.Main();
         return 0;
      }
   }

   class MathExample
   {
      static int Add(int x, int y)
      {
         return x + y;
      }

      public static int Main()
```

```
         {
            int i = Add(5,10);
            Console.WriteLine(i);
            return 0;
         }
      }
   }
```

code snippet DoubleMain.cs

This contains two classes, both of which have a `Main()` method. If you try to compile this code in the usual way, you will get the following errors:

```
csc DoubleMain.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.

DoubleMain.cs(7,25): error CS0017: Program
        'DoubleMain.exe' has more than one entry point defined:
        'Wrox.Client.Main()'.  Compile with /main to specify the type that
        contains the entry point.
DoubleMain.cs(21,25): error CS0017: Program
        'DoubleMain.exe' has more than one entry point defined:
        'Wrox.MathExample.Main()'.  Compile with /main to specify the type that
        contains the entry point.
```

However, you can explicitly tell the compiler which of these methods to use as the entry point for the program by using the `/main` switch, together with the full name (including namespace) of the class to which the `Main()` method belongs:

```
csc DoubleMain.cs /main:Wrox.MathExample
```

Passing Arguments to Main()

The examples so far have shown only the `Main()` method without any parameters. However, when the program is invoked, you can get the CLR to pass any command-line arguments to the program by including a parameter. This parameter is a string array, traditionally called `args` (although C# will accept any name). The program can use this array to access any options passed through the command line when the program is started.

The following sample, `ArgsExample.cs`, loops through the string array passed in to the `Main()` method and writes the value of each option to the console window:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class ArgsExample
   {
      public static int Main(string[] args)
      {
         for (int i = 0; i < args.Length; i++)
         {
            Console.WriteLine(args[i]);
         }
         return 0;
      }
   }
}
```

code snippet ArgsExample.cs

You can compile this as usual using the command line. When you run the compiled executable, you can pass in arguments after the name of the program, as shown in this example:

```
ArgsExample /a /b /c
/a
/b
/c
```

MORE ON COMPILING C# FILES

You have seen how to compile console applications using `csc.exe`, but what about other types of applications? What if you want to reference a class library? The full set of compilation options for the C# compiler is of course detailed in the MSDN documentation, but we list here the most important options.

To answer the first question, you can specify what type of file you want to create using the `/target` switch, often abbreviated as `/t`. This can be one of those shown in the following table.

OPTION	OUTPUT
`/t:exe`	A console application (the default)
`/t:library`	A class library with a manifest
`/t:module`	A component without a manifest
`/t:winexe`	A Windows application (without a console window)

If you want a nonexecutable file (such as a DLL) to be loadable by the .NET runtime, you must compile it as a library. If you compile a C# file as a module, no assembly will be created. Although modules cannot be loaded by the runtime, they can be compiled into another manifest using the `/addmodule` switch.

Another option we need to mention is `/out`. This allows you to specify the name of the output file produced by the compiler. If the `/out` option isn't specified, the compiler will base the name of the output file on the name of the input C# file, adding an extension according to the target type (for example, `exe` for a Windows or console application or `dll` for a class library). Note that the `/out` and `/t`, or `/target`, options must precede the name of the file you want to compile.

If you want to reference types in assemblies that aren't referenced by default, you can use the `/reference` or `/r` switch, together with the path and filename of the assembly. The following example demonstrates how you can compile a class library and then reference that library in another assembly. It consists of two files:

- The class library
- A console application, which will call a class in the library

The first file is called `MathLibrary.cs` and contains the code for your DLL. To keep things simple, it contains just one (public) class, `MathLib`, with a single method that adds two `int`s:

Available for download on Wrox.com

```
namespace Wrox
{
   public class MathLib
   {
      public int Add(int x, int y)
      {
         return x + y;
      }
   }
}
```

code snippet MathLibrary.cs

You can compile this C# file into a .NET DLL using the following command:

```
csc /t:library MathLibrary.cs
```

The console application, `MathClient.cs`, will simply instantiate this object and call its `Add()` method, displaying the result in the console window:

Available for download on Wrox.com

```
using System;

namespace Wrox
{
   class Client
   {
      public static void Main()
      {
         MathLib mathObj = new MathLib();
         Console.WriteLine(mathObj.Add(7,8));
      }
   }
}
```

code snippet MathClient.cs

You can compile this code using the `/r` switch to point at or reference the newly compiled DLL:

```
csc MathClient.cs /r:MathLibrary.dll
```

You can then run it as normal just by entering `MathClient` at the command prompt. This displays the number `15` — the result of your addition.

CONSOLE I/O

By this point, you should have a basic familiarity with C#'s data types, as well as some knowledge of how the thread-of-control moves through a program that manipulates those data types. In this chapter, you have also used several of the `Console` class's static methods used for reading and writing data. Because these methods are so useful when writing basic C# programs, this section quickly reviews them in more detail.

To read a line of text from the console window, you use the `Console.ReadLine()` method. This will read an input stream (terminated when the user presses the `Return` key) from the console window and return the input string. There are also two corresponding methods for writing to the console, which you have already used extensively:

- `Console.Write()` — Writes the specified value to the console window.
- `Console.WriteLine()` — This does the same, but adds a newline character at the end of the output.

Various forms (overloads) of these methods exist for all the predefined types (including `object`), so in most cases you don't have to convert values to strings before you display them.

For example, the following code lets the user input a line of text and displays that text:

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

`Console.WriteLine()` also allows you to display formatted output in a way comparable to C's `printf()` function. To use `WriteLine()` in this way, you pass in a number of parameters. The first is a string containing markers in curly braces where the subsequent parameters will be inserted into the text. Each marker contains a zero-based index for the number of the parameter in the following list. For example, `{0}` represents the first parameter in the list. Consider the following code:

```
int i = 10;
int j = 20;
Console.WriteLine("{0} plus {1} equals {2}", i, j, i + j);
```

This code displays:

```
10 plus 20 equals 30
```

You can also specify a width for the value, and justify the text within that width, using positive values for right-justification and negative values for left-justification. To do this, use the format {n,w}, where n is the parameter index and w is the width value:

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n —— \n {2,4}", i, j, i + j);
```

The result of this is:

```
  940
+  73
 ————
 1013
```

Finally, you can also add a format string, together with an optional precision value. It is not possible to give a complete list of potential format strings because, as you will see in Chapter 9, "Strings and Regular Expressions," you can define your own format strings. However, the main ones in use for the predefined types are shown in the following table.

STRING	DESCRIPTION
C	Local currency format.
D	Decimal format. Converts an integer to base 10, and pads with leading zeros if a precision specifier is given.
E	Scientific (exponential) format. The precision specifier sets the number of decimal places (6 by default). The case of the format string (e or E) determines the case of the exponential symbol.
F	Fixed-point format; the precision specifier controls the number of decimal places. Zero is acceptable.
G	General format. Uses E or F formatting, depending on which is more compact.
N	Number format. Formats the number with commas as the thousands separators, for example 32,767.44.
P	Percent format.
X	Hexadecimal format. The precision specifier can be used to pad with leading zeros.

Note that the format strings are normally case insensitive, except for e/E.

If you want to use a format string, you should place it immediately after the marker that gives the parameter number and field width, and separate it with a colon. For example, to format a decimal value as currency for the computer's locale, with precision to two decimal places, you would use C2:

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n —— \n {2,9:C2}", i, j, i + j);
```

The output of this in U.S. currency is:

```
   $940.23
+   $73.70
 —————————
 $1,013.93
```

As a final trick, you can also use placeholder characters instead of these format strings to map out formatting. For example:

```
double d = 0.234;
Console.WriteLine("{0:#.00}", d);
```

This displays as `.23`, because the `#` symbol is ignored if there is no character in that place, and zeros will either be replaced by the character in that position if there is one or be printed as a zero.

USING COMMENTS

The next topic — adding comments to your code — looks very simple on the surface, but can be complex.

Internal Comments within the Source Files

As noted earlier in this chapter, C# uses the traditional C-type single-line (`//...`) and multiline (`/* ... */`) comments:

```
// This is a singleline comment
/* This comment
   spans multiple lines. */
```

Everything in a single-line comment, from the `//` to the end of the line, will be ignored by the compiler, and everything from an opening `/*` to the next `*/` in a multiline comment combination will be ignored. Obviously, you can't include the combination `*/` in any multiline comments, because this will be treated as the end of the comment.

It is actually possible to put multiline comments within a line of code:

```
Console.WriteLine(/* Here's a comment! */ "This will compile.");
```

Use inline comments with care because they can make code hard to read. However, they can be useful when debugging if, say, you temporarily want to try running the code with a different value somewhere:

```
DoSomething(Width, /*Height*/ 100);
```

Comment characters included in string literals are, of course, treated like normal characters:

```
string s = "/* This is just a normal string .*/";
```

XML Documentation

In addition to the C-type comments, illustrated in the preceding section, C# has a very neat feature that we want to highlight: the ability to produce documentation in XML format automatically from special comments. These comments are single-line comments but begin with three slashes (`///`) instead of the usual two. Within these comments, you can place XML tags containing documentation of the types and type members in your code.

The tags in the following table are recognized by the compiler.

TAG	DESCRIPTION
`<c>`	Marks up text within a line as code, for example `<c>int i = 10;</c>`.
`<code>`	Marks multiple lines as code.
`<example>`	Marks up a code example.
`<exception>`	Documents an exception class. (Syntax is verified by the compiler.)
`<include>`	Includes comments from another documentation file. (Syntax is verified by the compiler.)
`<list>`	Inserts a list into the documentation.
`<param>`	Marks up a method parameter. (Syntax is verified by the compiler.)
`<paramref>`	Indicates that a word is a method parameter. (Syntax is verified by the compiler.)
`<permission>`	Documents access to a member. (Syntax is verified by the compiler.)

continues

(continued)

TAG	DESCRIPTION
`<remarks>`	Adds a description for a member.
`<returns>`	Documents the return value for a method.
`<see>`	Provides a cross-reference to another parameter. (Syntax is verified by the compiler.)
`<seealso>`	Provides a "see also" section in a description. (Syntax is verified by the compiler.)
`<summary>`	Provides a short summary of a type or member.
`<value>`	Describes a property.

To see how this works, add some XML comments to the `MathLibrary.cs` file from the previous "More on Compiling C# Files" section. You will add a `<summary>` element for the class and for its `Add()` method, and also a `<returns>` element and two `<param>` elements for the `Add()` method:

```
// MathLib.cs
namespace Wrox
{

   ///<summary>
   ///   Wrox.Math class.
   ///   Provides a method to add two integers.
   ///</summary>
   public class MathLib
   {
      ///<summary>
      ///   The Add method allows us to add two integers.
      ///</summary>
      ///<returns>Result of the addition (int)</returns>
      ///<param name="x">First number to add</param>
      ///<param name="y">Second number to add</param>
      public int Add(int x, int y)
      {
         return x + y;
      }
   }
}
```

The C# compiler can extract the XML elements from the special comments and use them to generate an XML file. To get the compiler to generate the XML documentation for an assembly, you specify the `/doc` option when you compile, together with the name of the file you want to be created:

```
csc /t:library /doc:MathLibrary.xml MathLibrary.cs
```

The compiler will throw an error if the XML comments do not result in a well-formed XML document.

This will generate an XML file named `Math.xml`, which looks like this:

Available for download on Wrox.com

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>MathLibrary</name>
   </assembly>
   <members>
      <member name="T:Wrox.MathLibrary">
         <summary>
            Wrox.MathLibrary class.
            Provides a method to add two integers.
         </summary>
      </member>
      <member name=
            "M:Wrox.MathLibrary.Add(System.Int32,System.Int32)">
            <summary>
               The Add method allows us to add two integers.
            </summary>
            <returns>Result of the addition (int)</returns>
            <param name="x">First number to add</param>
            <param name="y">Second number to add</param>
         </member>
      </members>
   </doc>
```

code snippet MathLibrary.xml

Notice how the compiler has actually done some work for you; it has created an `<assembly>` element and also added a `<member>` element for each type or member of a type in the file. Each `<member>` element has a `name` attribute with the full name of the member as its value, prefixed by a letter that indicates whether this is a type (`T:`), field (`F:`), or member (`M:`).

THE C# PREPROCESSOR DIRECTIVES

Besides the usual keywords, most of which you have now encountered, C# also includes a number of commands that are known as *preprocessor directives*. These commands never actually get translated to any commands in your executable code, but instead they affect aspects of the compilation process. For example, you can use preprocessor directives to prevent the compiler from compiling certain portions of your code. You might do this if you are planning to release two versions of the code — a basic version and an enterprise version that will have more features. You could use preprocessor directives to prevent the compiler from compiling code related to the additional features when you are compiling the basic version of the software. Another scenario is that you might have written bits of code that are intended to provide you with debugging information. You probably don't want those portions of code compiled when you actually ship the software.

The preprocessor directives are all distinguished by beginning with the `#` symbol.

C++ developers will recognize the preprocessor directives as something that plays an important part in C and C++. However, there aren't as many preprocessor directives in C#, and they are not used as often. C# provides other mechanisms, such as custom attributes, that achieve some of the same effects as C++ directives. Also, note that C# doesn't actually have a separate preprocessor in the way that C++ does. The so-called preprocessor directives are actually handled by the compiler. Nevertheless, C# retains the name preprocessor directive because these commands give the impression of a preprocessor.

The next sections briefly cover the purposes of the preprocessor directives.

#define and #undef

`#define` is used like this:

```
#define DEBUG
```

What this does is tell the compiler that a symbol with the given name (in this case `DEBUG`) exists. It is a little bit like declaring a variable, except that this variable doesn't really have a value — it just exists. And this symbol isn't part of your actual code; it exists only for the benefit of the compiler, while the compiler is compiling the code, and has no meaning within the C# code itself.

#undef does the opposite, and removes the definition of a symbol:

```
#undef DEBUG
```

If the symbol doesn't exist in the first place, then #undef has no effect. Similarly, #define has no effect if a symbol already exists.

You need to place any #define and #undef directives at the beginning of the C# source file, before any code that declares any objects to be compiled.

#define isn't much use on its own, but when combined with other preprocessor directives, especially #if, it becomes very powerful.

Incidentally, you might notice some changes from the usual C# syntax. Preprocessor directives are not terminated by semicolons and normally constitute the only command on a line. That's because for the preprocessor directives, C# abandons its usual practice of requiring commands to be separated by semicolons. If it sees a preprocessor directive, it assumes that the next command is on the next line.

#if, #elif, #else, and #endif

These directives inform the compiler whether to compile a block of code. Consider this method:

```
int DoSomeWork(double x)
{
   // do something
   #if DEBUG
      Console.WriteLine("x is " + x);
   #endif
}
```

This code will compile as normal, except for the Console.WriteLine() method call that is contained inside the #if clause. This line will be executed only if the symbol DEBUG has been defined by a previous #define directive. When the compiler finds the #if directive, it checks to see if the symbol concerned exists and compiles the code inside the #if clause only if the symbol does exist. Otherwise, the compiler simply ignores all the code until it reaches the matching #endif directive. Typical practice is to define the symbol DEBUG while you are debugging and have various bits of debugging-related code inside #if clauses. Then, when you are close to shipping, you simply comment out the #define directive, and all the debugging code miraculously disappears, the size of the executable file gets smaller, and your end users don't get confused by being shown debugging information. (Obviously, you would do more testing to make sure your code still works without DEBUG defined.) This technique is very common in C and C++ programming and is known as *conditional compilation*.

The #elif (=else if) and #else directives can be used in #if blocks and have intuitively obvious meanings. It is also possible to nest #if blocks:

```
#define ENTERPRISE
#define W2K

// further on in the file

#if ENTERPRISE
   // do something
   #if W2K
      // some code that is only relevant to enterprise
      // edition running on W2K
   #endif
#elif PROFESSIONAL
```

```
   // do something else
#else
   // code for the leaner version
#endif
```

Note that, unlike the situation in C++, using `#if` *is not the only way to compile code conditionally. C# provides an alternative mechanism through the* `Conditional` *attribute, which is explored in Chapter 14, "Reflection."*

`#if` and `#elif` support a limited range of logical operators too, using the operators `!`, `==`, `!=`, and `||`. A symbol is considered to be `true` if it exists and `false` if it doesn't. For example:

```
#if W2K && (ENTERPRISE==false)   // if W2K is defined but ENTERPRISE isn't
```

#warning and #error

Two other very useful preprocessor directives are `#warning` and `#error`. These will respectively cause a warning or an error to be raised when the compiler encounters them. If the compiler sees a `#warning` directive, it will display whatever text appears after the `#warning` to the user, after which compilation continues. If it encounters a `#error` directive, it will display the subsequent text to the user as if it were a compilation error message and then immediately abandon the compilation, so no IL code will be generated.

You can use these directives as checks that you haven't done anything silly with your `#define` statements; you can also use the `#warning` statements to remind yourself to do something:

```
#if DEBUG && RELEASE
   #error "You've defined DEBUG and RELEASE simultaneously!"
#endif

#warning "Don't forget to remove this line before the boss tests the code!"
   Console.WriteLine("*I hate this job.*");
```

#region and #endregion

The `#region` and `#endregion` directives are used to indicate that a certain block of code is to be treated as a single block with a given name, like this:

```
#region Member Field Declarations
   int x;
   double d;
   Currency balance;
#endregion
```

This doesn't look that useful by itself; it doesn't affect the compilation process in any way. However, the real advantage is that these directives are recognized by some editors, including the Visual Studio .NET editor. These editors can use these directives to lay out your code better on the screen. You will see how this works in Chapter 16.

#line

The `#line` directive can be used to alter the filename and line number information that is output by the compiler in warnings and error messages. You probably won't want to use this directive that often. It's most useful when you are coding in conjunction with some other package that alters the code you are typing in before sending it to the compiler. In this situation, line numbers, or perhaps the filenames reported by the compiler, won't match up to the line numbers in the files or the filenames you are editing. The `#line` directive can be used to restore the match. You can also use the syntax `#line default` to restore the line to the default line numbering:

```
#line 164 "Core.cs" // We happen to know this is line 164 in the file
                    // Core.cs, before the intermediate
                    // package mangles it.

// later on

#line default      // restores default line numbering
```

#pragma

The `#pragma` directive can either suppress or restore specific compiler warnings. Unlike command-line options, the `#pragma` directive can be implemented on a class or method level, allowing fine-grained control of what warnings are suppressed and when. The following example disables the "field not used" warning and then restores it after the `MyClass` class compiles:

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

C# PROGRAMMING GUIDELINES

The final section of this chapter supplies the guidelines you need to bear in mind when writing C# programs.

Rules for Identifiers

This section examines the rules governing what names you can use for variables, classes, methods, and so on. Note that the rules presented in this section are not merely guidelines: they are enforced by the C# compiler.

Identifiers are the names you give to variables, to user-defined types such as classes and structs, and to members of these types. Identifiers are case-sensitive, so, for example, variables named `interestRate` and `InterestRate` would be recognized as different variables. Following are a few rules determining what identifiers you can use in C#:

- They must begin with a letter or underscore, although they can contain numeric characters.
- You can't use C# keywords as identifiers.

The following table lists the C# reserved keywords.

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint

char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

If you need to use one of these words as an identifier (for example, if you are accessing a class written in a different language), you can prefix the identifier with the @ symbol to indicate to the compiler that what follows is to be treated as an identifier, not as a C# keyword (so `abstract` is not a valid identifier, but `@abstract` is).

Finally, identifiers can also contain Unicode characters, specified using the syntax `\uXXXX`, where `XXXX` is the four-digit hex code for the Unicode character. The following are some examples of valid identifiers:

- `Name`
- `Überfluß`
- `_Identifier`
- `\u005fIdentifier`

The last two items in this list are identical and interchangeable (because `005f` is the Unicode code for the underscore character), so obviously these identifiers couldn't both be declared in the same scope. Note that although syntactically you are allowed to use the underscore character in identifiers, this isn't recommended in most situations. That's because it doesn't follow the guidelines for naming variables that Microsoft has written to ensure that developers use the same conventions, making it easier to read each other's code.

Usage Conventions

In any development language, there usually arise certain traditional programming styles. The styles are not part of the language itself but are conventions concerning, for example, how variables are named or how certain classes, methods, or functions are used. If most developers using that language follow the same conventions, it makes it easier for different developers to understand each other's code — which in turn generally helps program maintainability. Conventions do, however, depend on the language and the environment. For example, C++ developers programming on the Windows platform have traditionally used the prefixes `psz` or `lpsz` to indicate strings — `char *pszResult; char *lpszMessage;` — but on Unix machines it's more common not to use any such prefixes: `char *Result; char *Message;`.

You'll notice from the sample code in this book that the convention in C# is to name variables without prefixes: `string Result; string Message;`.

The convention by which variable names are prefixed with letters that represent the data type is known as Hungarian notation. *It means that other developers reading the code can immediately tell from the variable name what data type the variable represents. Hungarian notation is widely regarded as redundant in these days of smart editors and IntelliSense.*

Whereas, with many languages, usage conventions simply evolved as the language was used, with C# and the whole of the .NET Framework, Microsoft has written very comprehensive usage guidelines, which are detailed in the .NET/C# MSDN documentation. This should mean that, right from the start, .NET programs will have a high degree of interoperability in terms of developers being able to understand code. The guidelines have also been developed with the benefit of some 20 years' hindsight in object-oriented programming. Judging by the relevant newsgroups, the guidelines have been carefully thought out and are well received in the developer community. Hence the guidelines are well worth following.

It should be noted, however, that the guidelines are not the same as language specifications. You should try to follow the guidelines when you can. Nevertheless, you won't run into problems if you have a good reason for not doing so — for example, you won't get a compilation error because you don't follow these guidelines. The general rule is that if you don't follow the usage guidelines you must have a convincing reason. Departing from the guidelines should be a positive decision rather than simply not bothering. Also, if you compare the guidelines with the samples in the remainder of this book, you'll notice that in numerous examples we have chosen not to follow the conventions. That's usually because the conventions are designed for much larger programs than our samples, and although they are great if you are writing a complete software package, they are not really suitable for small 20-line standalone programs. In many cases, following the conventions would have made our samples harder, rather than easier, to follow.

The full guidelines for good programming style are quite extensive. This section is confined to describing some of the more important guidelines, as well as the ones most likely to surprise you. If you want to make absolutely certain that your code follows the usage guidelines completely, you will need to refer to the MSDN documentation.

Naming Conventions

One important aspect to making your programs understandable is how you choose to name your items — and that includes naming variables, methods, classes, enumerations, and namespaces.

It is intuitively obvious that your names should reflect the purpose of the item and should not clash with other names. The general philosophy in the .NET Framework is also that the name of a variable should reflect the purpose of that variable instance and not the data type. For example, `height` is a good name for a variable, whereas `integerValue` isn't. However, you will probably feel that that principle is an ideal that is hard to achieve. Particularly when you are dealing with controls, in most cases, you'll probably be happier sticking with variable names such as `confirmationDialog` and `chooseEmployeeListBox`, which do indicate the data type in the name.

The following sections look at some of the things you need to think about when choosing names.

Casing of Names

In many cases you should use *Pascal casing* for names. Pascal casing means that the first letter of each word in a name is capitalized: `EmployeeSalary`, `ConfirmationDialog`, `PlainTextEncoding`. You will notice that essentially all of the names of namespaces, classes, and members in the base classes follow Pascal casing. In particular, the convention of joining words using the underscore character is discouraged. So, you should try not to use names such as `employee_salary`. It has also been common in other languages to use all capitals for names of constants. This is not advised in C# because such names are harder to read — the convention is to use Pascal casing throughout:

```
const int MaximumLength;
```

The only other casing scheme that you are advised to use is *camel casing*. Camel casing is similar to Pascal casing, except that the first letter of the first word in the name is not capitalized: `employeeSalary`, `confirmationDialog`, `plainTextEncoding`. Following are three situations in which you are advised to use camel casing:

- For names of all private member fields in types:

  ```
  public int subscriberId;
  ```

  Note, however, that often it is conventional to prefix names of member fields with an underscore:

  ```
  public int _subscriberId;
  ```

- For names of all parameters passed to methods:

  ```
  public void RecordSale(string salesmanName, int quantity);
  ```

- To distinguish items that would otherwise have the same name. A common example is when a property wraps around a field:

  ```
  private string employeeName;


  public string EmployeeName
  {
     get
     {
        return employeeName;

     }

  }
  ```

If you are doing this, you should always use camel casing for the private member and Pascal casing for the public or protected member, so that other classes that use your code see only names in Pascal case (except for parameter names).

You should also be wary about case sensitivity. C# is case-sensitive, so it is syntactically correct for names in C# to differ only by the case, as in the previous examples. However, you should bear in mind that your assemblies might at some point be called from Visual Basic .NET applications — and *Visual Basic .NET is not case-sensitive*. Hence, if you do use names that differ only by case, it is important to do so only in situations in which both names will never be seen outside your assembly. (The previous example qualifies as okay because camel case is used with the name that is attached to a `private` variable.) Otherwise, you may prevent other code written in Visual Basic .NET from being able to use your assembly correctly.

Name Styles

You should be consistent about your style of names. For example, if one of the methods in a class is called `ShowConfirmationDialog()`, then you should not give another method a name such as `ShowDialogWarning()` or `WarningDialogShow()`. The other method should be called `ShowWarningDialog()`.

Namespace Names

Namespace names are particularly important to design carefully to avoid risk of ending up with the same name for one of your namespaces as someone else's. Remember, namespace names are the *only* way that .NET distinguishes names of objects in shared assemblies. So, if you use the same namespace name for your software package as another package, and both packages get installed on the same computer, there are going to be problems. Because of this, it's almost always a good idea to create a top-level namespace with the name of your company and then nest successive namespaces that narrow down the technology, group, or department you are working in or the name of the package your classes are intended for. Microsoft recommends namespace names that begin with `<CompanyName>.<TechnologyName>` as in these two examples:

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

Names and Keywords

It is important that the names do not clash with any keywords. In fact, if you attempt to name an item in your code with a word that happens to be a C# keyword, you'll almost certainly get a syntax error because the compiler will assume that the name refers to a statement. However, because of the possibility that your classes will be accessed by code written in other languages, it is also important that you don't use names that are keywords in other .NET languages. Generally speaking, C++ keywords are similar to C# keywords, so confusion with C++ is unlikely, and those commonly encountered keywords that are unique to Visual C++ tend to start with two underscore characters. As with C#, C++ keywords are spelled in lowercase, so if you hold to the convention of naming your public classes and members with Pascal-style names, they will always have at least one uppercase letter in their names, and there will be no risk of clashes with C++ keywords. However, you are more likely to have problems with Visual Basic .NET, which has many more keywords than C# does, and being non-case–sensitive means that you cannot rely on Pascal-style names for your classes and methods.

The following table lists the keywords and standard function calls in Visual Basic .NET, which you should avoid, if possible, in whatever case combination, for your public C# classes.

Abs	Do	Loc	RGB
Add	Double	Local	Right
AddHandler	Each	Lock	RmDir
AddressOf	Else	LOF	Rnd
Alias	ElseIf	Log	RTrim
And	Empty	Long	SaveSettings
Ansi	End	Loop	Second
AppActivate	Enum	LTrim	Seek
Append	EOF	Me	Select
As	Erase	Mid	SetAttr
Asc	Err	Minute	SetException
Assembly	Error	MIRR	Shared
Atan	Event	MkDir	Shell
Auto	Exit	Module	Short
Beep	Exp	Month	Sign
Binary	Explicit	MustInherit	Sin
BitAnd	ExternalSource	MustOverride	Single
BitNot	False	MyBase	SLN
BitOr	FileAttr	MyClass	Space
BitXor	FileCopy	Namespace	Spc
Boolean	FileDateTime	New	Split
ByRef	FileLen	Next	Sqrt
Byte	Filter	Not	Static
ByVal	Finally	Nothing	Step
Call	Fix	NotInheritable	Stop
Case	For	NotOverridable	Str
Catch	Format	Now	StrComp

CBool	FreeFile	NPer	StrConv
CByte	Friend	NPV	Strict
CDate	Function	Null	String
CDbl	FV	Object	Structure
CDec	Get	Oct	Sub
ChDir	GetAllSettings	Off	Switch
ChDrive	GetAttr	On	SYD
Choose	GetException	Open	SyncLock
Chr	GetObject	Option	Tab
CInt	GetSetting	Optional	Tan
Class	GetType	Or	Text
Clear	GoTo	Overloads	Then
CLng	Handles	Overridable	Throw
Close	Hex	Overrides	TimeOfDay
Collection	Hour	ParamArray	Timer
Command	If	Pmt	TimeSerial
Compare	Iif	PPmt	TimeValue
Const	Implements	Preserve	To
Const	Implements	Preserve	To
Cos	Imports	Print	Today
CreateObject	In	Private	Trim
CShort	Inherits	Property	Try
CSng	Input	Public	TypeName
CStr	InStr	Put	TypeOf
CurDir	Int	PV	UBound
Date	Integer	QBColor	UCase
DateAdd	Interface	Raise	Unicode
DateDiff	Ipmt	RaiseEvent	Unlock
DatePart	IRR	Randomize	Until
DateSerial	Is	Rate	Val
DateValue	IsArray	Read	Weekday
Day	IsDate	ReadOnly	While
DDB	IsDbNull	ReDim	Width
Decimal	IsNumeric	Remove	With
Declare	Item	RemoveHandler	WithEvents
Default	Kill	Rename	Write
Delegate	Lcase	Replace	WriteOnly
DeleteSetting	Left	Reset	Xor
Dim	Lib	Resume	Year

Use of Properties and Methods

One area that can cause confusion in a class is whether a particular quantity should be represented by a property or a method. The rules here are not hard and fast, but in general, you ought to use a property if something really should look and feel like a variable. (If you're not sure what a property is, see Chapter 3.) This means, among other things, that:

- Client code should be able to read its value. Write-only properties are not recommended, so, for example, use a `SetPassword()` method, not a write-only `Password` property.
- Reading the value should not take too long. The fact that something is a property usually suggests that reading it will be relatively quick.
- Reading the value should not have any observable and unexpected side effect. Further, setting the value of a property should not have any side effect that is not directly related to the property. Setting the width of a dialog box has the obvious effect of changing the appearance of the dialog box on the screen. That's fine, because that's obviously related to the property in question.
- It should be possible to set properties in any order. In particular, it is not good practice when setting a property to throw an exception because another related property has not yet been set. For example, to use a class that accesses a database, you need to set `ConnectionString`, `UserName`, and `Password`, then the author of the class should make sure the class is implemented so that the user really can set them in any order.
- Successive reads of a property should give the same result. If the value of a property is likely to change unpredictably, you should code it as a method instead. `Speed`, in a class that monitors the motion of an automobile, is not a good candidate for a property. Use a `GetSpeed()` method here; but, `Weight` and `EngineSize` are good candidates for properties because they will not change for a given object.

If the item you are coding satisfies all the preceding criteria, it is probably a good candidate for a property. Otherwise, you should use a method.

Use of Fields

The guidelines are pretty simple here. Fields should almost always be private, except that in some cases it may be acceptable for constant or read-only fields to be public. The reason is that if you make a field public, you may hinder your ability to extend or modify the class in the future.

The previous guidelines should give you a foundation of good practices, and you should also use them in conjunction with a good object-oriented programming style.

A final helpful note to keep in mind is that Microsoft has been fairly careful about being consistent and has followed its own guidelines when writing the .NET base classes. So a very good way to get an intuitive feel for the conventions to follow when writing .NET code is to simply look at the base classes — see how classes, members, and namespaces are named, and how the class hierarchy works. Consistency between the base classes and your classes will help in readability and maintainability.

SUMMARY

This chapter examined some of the basic syntax of C#, covering the areas needed to write simple C# programs. We covered a lot of ground, but much of it will be instantly recognizable to developers who are familiar with any C-style language (or even JavaScript).

You have seen that although C# syntax is similar to C++ and Java syntax, there are many minor differences. You have also seen that in many areas this syntax is combined with facilities to write code very quickly, for example high-quality string handling facilities. C# also has a strongly defined type system, based on a distinction between value and reference types. Chapters 3 and 4 cover the C# object-oriented programming features.