CHAPTER 3

C# Language Fundamentals

Chapter 1 demonstrates a very simple C# program that prints the text string "Hello world!" to the console screen and provides a line-by-line analysis of that program. However, even that very simple program was complex enough that some of the details had to be skipped over. In this chapter, I'll begin an in-depth exploration of the syntax and structure of the C# language. The *syntax* of a language is the order of the keywords, where you put semicolons and so forth. The *semantics* is what you are expressing in the code, and how your code fits together. Syntax is trivial and unimportant, but because compilers are absolute sticklers for correct syntax, novice programmers spend a lot of attention to syntax until they are comfortable. Fortunately, Visual Studio 2005 makes managing syntax much easier so that you can focus on semantics, which is far more important.

In this chapter, I'll introduce statements and expressions, the building blocks of any program. You'll learn about variables and constants, which let you store values for use in your program. I'll begin an explanation of types and we'll take a look at strings, which you saw briefly in the Hello World program.

Statements

In C#, a complete program instruction is called a *statement* and each statement ends with a semicolon (;). Programs consist of sequences of statements such as:

```
int myVariable;                  // a statement
myVariable = 23;                 // another statement
int anotherVariable = myVariable; // yet another statement
```

The compiler starts at the beginning of a source code file and reads down, executing statement after statement in the order encountered. This would be entirely straightforward, and terribly limiting, were it not for branching. Branching allows you to change the order in which statements are evaluated. See Chapter 5 for more information about branching.

Types

C# is a *strongly typed* language. That means that every object you create or use in a C# program must have a specific *type* (e.g., you must declare the object to be an integer or a string or a Dog or a Button). Essentially, the type indicates how big the object is (in memory) and what it can do.

Types come in two flavors: those that are built into the language (intrinsic types) and those you create (classes and interfaces, discussed in Chapters 7 and 13). C# offers a number of intrinsic types, shown in Table 3-1.

Table 3-1. The intrinsic types

C# type	Size (in bytes)	.NET type	Description
`byte`	1	`Byte`	Unsigned (values 0–255).
`char`	2	`Char`	Unicode characters.
`bool`	1	`Boolean`	True or false.
`sbyte`	1	`SByte`	Signed (values –128 to 127).
`short`	2	`Int16`	Signed (short) (values –32,768 to 32,767).
`ushort`	2	`UInt16`	Unsigned (short) (values 0 to 65,535).
`int`	4	`Int32`	Signed integer values between –2,147,483,648 and 2,147,483,647.
`uint`	4	`UInt32`	Unsigned integer values between 0 and 4,294,967,295.
`float`	4	`Single`	Floating point number. Holds the values from approximately +/– $1.5 * 10^{-45}$ to approximately +/–$3.4 * 10^{38}$ with 7 significant figures.
`double`	8	`Double`	Double-precision floating point; holds the values from approximately +/-$5.0 * 10^{-324}$ to approximately +/–$1.8 * 10^{308}$ with 15–16 significant figures.
`decimal`	12	`Decimal`	Fixed-precision up to 28 digits and the position of the decimal point. This is typically used in financial calculations. Requires the suffix "m" or "M."
`long`	8	`Int64`	Signed integers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`ulong`	8	`UInt64`	Unsigned integers ranging from 0 to approximately $1.85 * 10^{19}$.

Each type has a name (such as `int`) and a size (such as 4 bytes). The size tells you how many bytes each object of this type occupies in memory. (Programmers generally don't like to waste memory if they can avoid it, but with the cost of memory these days, you can afford to be mildly profligate if doing so simplifies your program.) The description field of Table 3-1 tells you the minimum and maximum values you can hold in objects of each type.

Each C# type corresponds to an underlying .NET type. Thus, what C# calls an `int`, .NET calls an `Int32`. This is interesting only if you care about sharing objects across languages.

Intrinsic types can't do much. You can use them to add two numbers together, and they can display their values as strings. User-defined types can do a lot more; their abilities are determined by the methods you create, as discussed in detail in Chapter 8.

Objects of an intrinsic type are called *variables*. Variables are discussed in detail later in this chapter.

Numeric Types

Most of the intrinsic types are used for working with numeric values (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `float`, `double`, `decimal`, `long`, and `ulong`).

The numeric types can be broken into two sets: unsigned and signed. An unsigned value (`byte`, `ushort`, `uint`, `ulong`) can hold only positive values. A signed value (`sbyte`, `short`, `int`, `long`) can hold positive or negative values, but the highest value is only half as large as the corresponding unsigned type. That is, a `ushort` can hold any value from 0 through 65,535, but a `short` can hold only –32,768 through 32,767. Notice that 32,767 is nearly half of 65,535 (it is off by one to allow for holding the value zero). The reason a `ushort` can hold up to 65,535 is that 65,536 is a round number in binary arithmetic (2^{16}) and one bit is devoted to 0.

Another way to categorize the types is into those used for integer values (whole numbers) and those used for floating-point values (fractional or rational numbers). The `byte`, `sbyte`, `ushort`, `uint`, `ulong`, `short`, `int`, and `long` types all hold whole number values.

The `byte` and `sbyte` types are not used very often and won't be described in this book.

The `double` and `float` types hold fractional values. For most uses, `float` will suffice, unless you need to hold a really big fractional number, in which case you might use a `double`. The `decimal` value type was added to the language to support scientific and financial applications.

Typically, you decide which size integer to use (`short`, `int`, or `long`) based on the magnitude of the value you want to store. For example, a `ushort` can only hold values from 0 through 65,535, while a `uint` can hold values from 0 through 4,294,967,295.

That said, in real life, most of the time you'll simply declare your numeric variables to be of type `int`, unless there is a good reason to do otherwise. (Most programmers choose signed types unless they have a good reason to use an unsigned value. This is, in part, just a matter of tradition.)

Suppose you need to keep track of inventory for a book warehouse. You expect to house up to 40,000 or even 50,000 copies of each book. A signed `short` can only hold up to 32,767 values. You might be tempted to use an unsigned `short` (which can hold up to 65,535 values), but it is easier and preferable to just use a signed `int` (with a maximum value of 2,147,483,647). That way, if you have a runaway best seller, your program won't break (if you anticipate selling more than 2 billion copies of your book, perhaps you'll want to use a `long`!).

Throughout this book, we will use `int` wherever it works, even if `short` or `byte` might be workable alternatives. Memory is cheap, and programmer time expensive. There are circumstances where the difference in memory usage would be significant (for example, if you are going to hold a billion of them in memory), but we'll keep things simple by using the `int` type whenever possible.

`float`, `double`, and `decimal` offer varying degrees of size and precision. For most small fractional numbers, `float` is fine. Note that the compiler assumes that any number with a decimal point is a `double` unless you tell it otherwise. (The "Variables"" section discusses how you tell it otherwise.)

Non-Numeric Types: char and bool

In addition to the numeric types, the C# language offers two other types: `char` and `bool`.

The `char` type is used from time to time when you need to hold a single character. The `char` type can represent a simple character (`A`), a Unicode character (`\u0041`), or an escape sequence (`'\n'`). You'll see escape sequences later in this book, and their use will be explained in context.

The one remaining important type is `bool`, which holds a Boolean value. A Boolean value is one that is either true or false. Boolean values are used frequently in C# programming, as you'll see throughout this book. Virtually every comparison (is `myDog` bigger than `yourDog`?) results in a Boolean value.

The `bool` type was named after George Boole (1815–1864), an English mathematician who published *An Investigation into the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities,* and thus created the science of Boolean algebra.

Types and Compiler Errors

The compiler will help you by complaining if you try to use a type improperly. The compiler complains in one of two ways: it issues a warning or it issues an error.

You are well advised to treat warnings as errors. Stop what you are doing and figure out why there is a warning and fix the problem. Never ignore a compiler warning unless you are certain that you know exactly why the warning was issued and that you know something the compiler does not.

To have Visual Studio enforce this for you, follow these steps:

1. Right-click on the project.
2. Click on the Compile tab.
3. Make sure the "Treat all warnings as errors" checkbox is checked or set the Warnings that you want to treat as errors using the drop-down boxes.

Programmers talk about design-time, compile-time, and runtime. Design-time is when you are designing the program, compile-time is when you compile the program, and runtime is (surprise!) when you run the program.

The earlier in your development process that you unearth a bug, the better. It is easier to fix a bug in your logic at design-time than to fix the bug once it has been written into code. Likewise, it is better (and cheaper) to find bugs in your program at compile-time than at runtime. Not only is it better; it is more reliable. A compile-time bug will fail every time you run the compiler, but a runtime bug can hide. Runtime bugs slip under a crack in your logic and lurk there (sometimes for months), biding their time, waiting to come out when it will be most expensive (or most embarrassing) to you.

It will be a constant theme of this book that you *want* the compiler to find bugs. The compiler is your friend (though I admit, at times it feels like your Nemesis). The more bugs the compiler finds, the fewer bugs your users will find.

A strongly typed language like C# helps the compiler find bugs in your code. Here's how: suppose you tell the compiler that Milo is of type `Dog`. Sometime later you try to use Milo to display text (calling the `ShowText` method). Oops, `Dogs` don't display text. Your compiler will stop with an error:

```
Dog does not contain a definition for 'showText'
```

Very nice. Now you can go figure out if you used the wrong object or you called the wrong method.

Visual Studio .NET actually finds the error even before the compiler does. When you try to add a method, IntelliSense pops up a list of valid methods to help you, as shown in Figure 3-1.

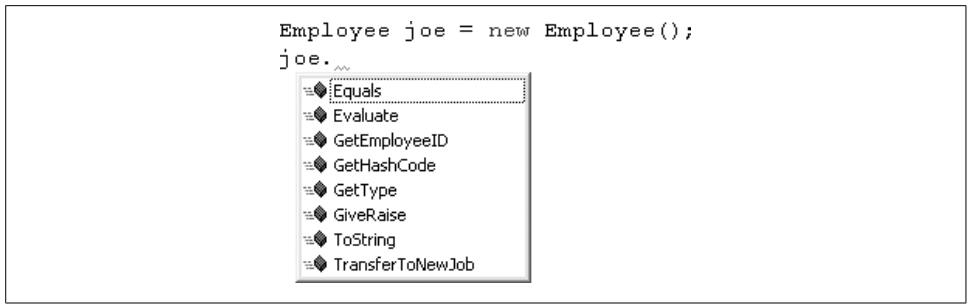

Figure 3-1. IntelliSense

When you try to add a method that does not exist, it won't be in the list. That is a pretty good clue that you are not using the object properly.

Variables

A variable is an instance of an intrinsic type (such as `int`) that can hold a value:

```
int myVariable = 15;
```

You *initialize* a variable by writing its type, its *identifier*, and then assigning a value to that variable.

An *identifier* is just an arbitrary name you assign to a variable, method, class, or other element. In this case, the variable's identifier is `myVariable`.

You can define variables without initializing them:

```
int myVariable;
```

You can then assign a value to `myVariable` later in your program:

```
int myVariable;
// some other code here
myVariable = 15; // assign 15 to myVariable
```

You can also change the value of a variable later in the program. That is why they're called variables; their values can vary.

```
int myVariable;
// some other code here
myVariable = 15; // assign 15 to myVariable
// some other code here
myVariable = 12; // now it is 12
```

Technically, a variable is a named storage location (that is, stored in memory) with a type. After the final line of code in the previous example, the value 12 is stored in the named location `myVariable`.

Example 3-1 illustrates the use of variables. To test this program, open Visual Studio .NET and create a console application. Type in the code as shown.

WriteLine( )

The .NET Framework provides a useful method for displaying output on the screen in console applications: `System.Console.WriteLine( )`. How you use this method will become clearer as you progress through the book, but the fundamentals are straightforward. You call the method, passing in a string that you want printed to the console (the screen), as in the Hello World application in Chapter 1.

You can also pass in substitution parameters. A *substitution parameter* is just a placeholder for a value you want to display. For example, you might pass in the substitution parameter {0}, and then when you run the program, you'll substitute the value held in the variable `myInt`, so that its value is displayed where the parameter {0} appears in the `WriteLine( )` statement.

Here's how it works. You place a number between braces:

```
System.Console.WriteLine("After assignment, myInt: {0}", myInt);
```

Notice that you follow the quoted string with a comma and then a variable name. The value of the variable will be substituted into the parameter. Assuming `myInt` has the value 15, the statement shown previously causes the following to display:

```
After assignment, myInt: 15
```

If you have more than one parameter, the variable values will be substituted in order, as in the following:

```
System.Console.WriteLine("After assignment, myInt: {0} and
myOtherInt: {1}", myInt, myOtherInt);
```

Assuming `myInt` has the value 15, and `myOtherInt` has the value 20, this will cause the following to display:

```
After assignment, myInt: 15 and myOtherInt: 20.
```

You'll see a great deal more about `WriteLine( )` in later chapters.

Example 3-1. Using variables

```
class Values
{
     static void Main( )
     {
          int myInt = 7;
          System.Console.WriteLine("Initialized, myInt: {0}",
          myInt);
          myInt = 5;
          System.Console.WriteLine("After assignment, myInt: {0}",
          myInt);
     }
}
```

Press F5 to build and run this application; the output looks like this:

```
Initialized, myInt: 7
After assignment, myInt: 5
```

Example 3-1 initializes the variable `myInt` to the value 7, displays that value, reassigns the variable with the value 5, and displays it again.

Definite Assignment

C# requires *definite assignment*; that is, variables must be initialized (or assigned to) before they are "used" (see "out Parameters and Definite Assignment" in Chapter 8). To test this rule, change the line that initializes `myInt` in Example 3-1 to:

```
int myInt;
```

Save the revised program shown in Example 3-2.

Example 3-2. Uninitialized variable

```
class Values
{
     static void Main( )
     {
         int myInt;
         System.Console.WriteLine
         ("Uninitialized, myInt: {0}",myInt);
         myInt = 5;
         System.Console.WriteLine("Assigned, myInt: {0}", myInt);
     }
}
```

When you try to compile Example 3-2, the C# compiler will display the following error message:

```
Use of unassigned local variable 'myInt'
```

It is not legal to use an uninitialized variable in C#; doing so violates the rule of definite assignment. In this case, "using" the variable `myInt` means passing it to `WriteLine( )`.

So does this mean you must initialize every variable? No, but if you don't initialize your variable, then you must assign a value to it before you attempt to use it. Example 3-3 illustrates a corrected program.

Example 3-3. Definite assignment

```
class Values
{
     static void Main( )
     {
         int myInt;
         //other code here...
         myInt = 7; // assign to it
```

Example 3-3. Definite assignment (continued)

```
        System.Console.WriteLine("Assigned, myInt: {0}", myInt);
        myInt = 5;
        System.Console.WriteLine("Reassigned, myInt: {0}", myInt);
    }
}
```

Constants

Variables are a powerful tool, but there are times when you want to manipulate a defined value, one whose value you want to ensure remains constant. A *constant* is like a variable in that it can store a value. However, unlike a variable, you cannot change the value of a constant while the program runs.

For example, you might need to work with the Fahrenheit freezing and boiling points of water in a program simulating a chemistry experiment. Your program will be clearer if you name the variables that store these values `FreezingPoint` and `BoilingPoint`, but you do not want to permit their values to be changed while the program is executing. The solution is to use a constant. Constants come in three flavors: *literals*, *symbolic constants*, and *enumerations*.

Literal Constants

A literal constant is just a value. For example, 32 is a literal constant. It does not have a name; it is just a literal value. And you can't make the value 32 represent any other value. The value of 32 is always 32. You can't assign a new value to 32, and you can't make 32 represent the value 99 no matter how hard you might try.

Symbolic Constants

Symbolic constants assign a name to a constant value. You declare a symbolic constant using the following syntax:

```
const type identifier = value;
```

The `const` keyword is followed by a type, an identifier, the assignment operator (`=`), and the value with which you'll initialize the constant.

This is similar to declaring a variable, except that you start with the keyword `const` and symbolic constants *must* be initialized. Once initialized, a symbolic constant cannot be altered. For example, in the following declaration, 32 is a literal constant and `FreezingPoint` is a symbolic constant of type `int`:

```
const int FreezingPoint = 32;
```

Example 3-4 illustrates the use of symbolic constants.

Example 3-4. Using symbolic constants

```
class Values
{
 static void Main( )
 {
 const int FreezingPoint = 32; // degrees Fahrenheit
 const int BoilingPoint = 212;

 System.Console.WriteLine("Freezing point of water: {0}",
 FreezingPoint );
 System.Console.WriteLine("Boiling point of water: {0}",
 BoilingPoint );
 //BoilingPoint = 21;

 }
}
```

Example 3-4 creates two symbolic integer constants: `FreezingPoint` and `BoilingPoint`. See the sidebar, "Naming Conventions," for a discussion of how to name symbolic constants.

Naming Conventions

Microsoft has promulgated white papers on how you should name the variables, constants, and other objects in your program. They define two types of naming conventions: Camel notation and Pascal notation.

In Camel notation, names begin with a lowercase letter. Multiword names (such as "my button") are written with no spaces and no underscore and with each word after the first capitalized. Thus, the correct name for "my button" is myButton.

Pascal notation is just like Camel notation except that the first letter is also uppercase (FreezingPoint).

Microsoft suggests that variables be written with Camel notation and constants with Pascal notation. In later chapters, you'll learn that member variables are named using Camel notation, while methods and classes are named using Pascal notation.

These constants serve the same purpose as using the literal values 32 and 212 for the freezing and boiling points of water, respectively, in expressions that require them. However, because the constants have names, they convey far more meaning. It might seem easier to just use the literal values 32 and 212 instead of going to the trouble of declaring the constants, but if you decide to switch this program to Celsius, you can reinitialize these constants at compile time to 0 and 100, respectively, and all the rest of the code should continue to work.

To prove to yourself that the constant cannot be reassigned, try un-commenting the last line of the preceding program by removing the two slash marks:

```
BoilingPoint = 21;
```

When you recompile, you receive this error:

```
The left-hand side of an assignment must be a variable, property or indexer
```

Enumerations

Enumerations provide a powerful alternative to literal or simple symbolic constants. An *enumeration* is a distinct value type, consisting of a set of named constants (called the enumerator list).

In Example 3-4, you created two related constants:

```
const int FreezingPoint = 32;
const int BoilingPoint = 212;
```

You might want to add a number of other useful constants to this list as well, such as:

```
const int LightJacketWeather = 60;
const int SwimmingWeather = 72;
const int WickedCold = 0;
```

Notice, however, that this process is somewhat cumbersome; also, this syntax shows no logical connection among these various constants. C# provides an alternate construct, the *enumeration*, which allows you to group logically related constants, as in the following:

```
enum Temperatures
{
    WickedCold = 0,
    FreezingPoint = 32,
    LightJacketWeather = 60,
    SwimmingWeather = 72,
    BoilingPoint = 212,
}
```

Many programmers like to leave a comma after the last entry in an enumeration as a convenience for adding more values later. Other programmers find this, at best, sloppy. The code will compile either way.

The complete syntax for specifying an enumeration uses the `enum` keyword, as follows:

```
[attributes] [modifiers] enum identifier
[:base-type] {enumerator-list};
```

In a specification statement like the preceding example, anything in square brackets is optional. Thus, you can declare an `enum` with no attributes, modifiers, or base-type.

The optional attributes and modifiers are considered later in this book. For now, let's focus on the rest of this declaration. An enumeration begins with the keyword enum, which is generally followed by an identifier; in this case, Temperatures:

```
enum Temperatures
```

The base-type is the underlying type for the enumeration. You might specify that you are declaring constant ints, constant longs, or something else. If you leave out this optional value (and often you will), it defaults to int, but you are free to use any of the integral types (ushort, long) except for char. For example, the following fragment declares an enumeration with unsigned integers (uint) as the base-type:

```
enum ServingSizes : uint
{
     Small = 1,
     Regular = 2,
     Large = 3
}
```

Notice that an enum declaration ends with the enumerator list, which contains the constant assignments for the enumeration, each separated by a comma. Example 3-5 rewrites Example 3-4 to use an enumeration.

Example 3-5. Using an enumeration

```
class Values
{
     // declare the enumeration
     enum Temperatures
     {
         WickedCold = 0,
         FreezingPoint = 32,
         LightJacketWeather = 60,
         SwimmingWeather = 72,
         BoilingPoint = 212,
     }

     static void Main( )
     {

         System.Console.WriteLine("Freezing point of water: {0}",
         (int) Temperatures.FreezingPoint );
         System.Console.WriteLine("Boiling point of water: {0}",
         (int) Temperatures.BoilingPoint );
     }
}
```

In Example 3-5, you declare an enumerated constant called Temperatures. When you want to use any of the values in an enumeration in a program, the values of the enumeration must be qualified by the enumeration name.

You cannot just refer to FreezingPoint; instead, you use the enumeration identifier (Temperature) followed by the dot operator and then the enumerated constant

(FreezingPoint). This is called *qualifying* the identifier FreezingPoint. Thus, to refer to the FreezingPoint, you use the full identifier Temperature.FreezingPoint.

You might want to display the value of an enumerated constant to the console, as in the following:

```
Console.WriteLine("The freezing point of water is {0}",
 (int) Temperature.FreezingPoint);
```

To make this work properly, you must cast the constant to its underlying type (int). When you *cast* a value, you tell the compiler "I know that this value is really of the indicated type." In this case, you are saying, "Treat this enumerated constant as an int. " Because the underlying type is int, this is safe to do. (See the sidebar, "Casting.")

Casting

Objects of one type can be converted into objects of another type. This is called *casting*. Casting can be either implicit or explicit.

An *implicit conversion* happens automatically; the compiler takes care of it for you. If you have a short, and you assign it to a variable of type int, the compiler automatically (and silently) casts it for you. You don't have to take any action. This is safe, because an int variable can hold any value that might have been in a short variable.

```
short myShort = 5;
// other code here...
int myint = myShort; // implicit conversion
```

You use *explicit conversions* when there is danger of losing data. For example, while the compiler will let you convert a short to an int implicitly (no chance you can lose data), it will not let you implicitly convert an int to a short (the short may not be able to hold all the information in the integer). To accomplish this, you must use an explicit conversion—a signal to the compiler that you want the conversion even though it is dangerous. You do so by placing the type you want to convert to in parentheses:

```
int myInt = 5;
// other code here...
short myShort = (short) myInt; // explicit conversion
```

Note the (int)—that is the explicit conversion.

The semantics of an explicit conversion are "Hey! Compiler! I know what I'm doing." This is sometimes called "hitting it with the big hammer" and can be very useful or very painful, depending on whether your thumb is in the way.

In Example 3-5, the values in the two enumerated constants FreezingPoint and BoilingPoint are both cast to type integer; then that integer value is passed to WriteLine( ) and displayed.

Each constant in an enumeration corresponds to a numerical value. In Example 3-5, each enumerated value is an integer. If you don't specifically set it otherwise, the enumeration begins at 0 and each subsequent value counts up from the previous. Thus, if you create the following enumeration:

```
enum SomeValues
{
     First,
     Second,
     Third = 20,
     Fourth
}
```

the value of `First` will be 0, `Second` will be 1, `Third` will be 20, and `Fourth` will be 21.

Strings

It is nearly impossible to write a C# program without creating strings. A `string` object holds a series of characters.

You declare a string variable using the `string` keyword much as you would create an instance of any type:

```
string myString;
```

You specify a *string literal* by enclosing it in double quotes:

```
"Hello World"
```

It is common to initialize a string variable that contains a string literal:

```
string myString = "Hello World";
```

Strings will be covered in much greater detail in Chapter 15.

Expressions

Statements that evaluate to a value are called *expressions*. You may be surprised how many statements do evaluate to a value. For example, an assignment such as:

```
myVariable = 57;
```

is an expression; it evaluates to the value assigned—in this case, 57.

Note that the preceding statement assigns the value 57 to the variable `myVariable`. The assignment operator (=) does not test equality; rather, it causes whatever is on the right side (57) to be assigned to whatever is on the left side (`myVariable`). Chapter 4 discusses some of the more useful C# operators (including assignment and equality).

Because `myVariable = 57` is an expression that evaluates to 57, it can be used as part of another assignment, such as:

```
mySecondVariable = myVariable = 57;
```

What happens in this statement is that the literal value 57 is assigned to the variable `myVariable`. The value of that assignment (57) is then assigned to the second variable, `mySecondVariable`. Thus, the value 57 is assigned to both variables. You can assign a value to any number of variables with one statement using the assignment operator (=), as in the following:

```
int a,b,c,d,e;
a = b = c = d = e = 20;
```

Whitespace

In the C# language, spaces, tabs, and newlines are considered to be *whitespace* (so named because you see only the white of the underlying "page"). Extra whitespace is generally ignored in C# statements. Thus, you can write:

```
myVariable = 5;
```

or:

```
myVariable           =              5     ;
```

and the compiler will treat the two statements as identical. The key is to use whitespace to make the program more readable to the programmer; the compiler is indifferent.

The exception to this rule is that whitespace within a string is treated as literal; it is not ignored. If you write:

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

each space between "Hello" and "World" is treated as another character in the string. (In this case, there is only one space character.)

Problems arise only when you do not leave space between logical program elements that require it. For instance, the expression:

```
int                 myVariable     =               5          ;
```

is the same as:

```
int myVariable=5;
```

but it is *not* the same as:

```
intmyVariable =5;
```

The compiler knows that the whitespace on either side of the assignment operator is extra, but at least some whitespace between the type declaration `int` and the variable name `myVariable` is *not* extra; it is required.

This is not surprising; the whitespace allows the compiler to *parse* the keyword `int` rather than some unknown term `intmyVariable`. You are free to add as much or as little whitespace between `int` and `myVariable` as you care to, but there must be at least one whitespace character (typically a space or tab).

Visual Basic programmers take note: in C#, the end-of-line has no special significance. Statements are ended with semicolons, not newline characters. There is no line continuation character because none is needed.

Summary

- A complete program instruction is called a statement. Each statement ends with a semicolon (;).
- All objects, constants, and variables must have a specific type.
- Most of the intrinsic types are used for working with numeric values. You will commonly use `int` for whole numbers and `double` or `float` for fractional values.
- The `char` type is used for holding a single character.
- The `bool` type can only hold the value true or false.
- A variable is an instance of an intrinsic type. You initialize a variable by creating it with an assigned value.
- A constant is similar to a variable, but the value cannot be changed while the program is running.
- An enumeration is a value type that consists of a set of named constants.
- You can cast a value from one type to another as long as either the compiler knows how to turn the original type into the cast-to type, or you provide a method in your class definition that tells the compiler how to make the cast.
- If no information can be lost, you may cast from one type to another implicitly.
- If information may be lost (such as when casting from a long to an integer), you must cast explicitly.
- A string object holds a series of characters (such as a word or sentence).
- String objects are immutable; when you appear to be changing a string's value, you are actually creating a new string.
- Expressions are statements that evaluate to a value.
- Extra whitespace (spaces, tabs, and newline characters) that is not within a string, is ignored by the compiler.

Quiz

Question 3-1. What values can a `bool` type have?

Question 3-2. What is the difference between an `int` and an `Int32`?

Question 3-3. Which of the following code statements will compile?

```
int myInt = 25;
long myLong = myInt;
int newInt = myLong;
```

Question 3-4. What is the difference between an `int` and a `uint`?

Question 3-5. What is the difference between a `float` and a `double`?

Question 3-6. Explain definite assignment.

Question 3-7. Given the following declaration, how would you refer to the constant for `LightJacketWeather` and what would its value be?

```
enum Temperatures
{
    WickedCold = 0,
    FreezingPoint = 32,
    LightJacketWeather,
    SwimmingWeather = 72,
    BoilingPoint = 212,
}
```

Question 3-8. What is an expression?

Exercises

Exercise 3-1. Write a short program creating and initializing each of the following types of variables: `int`, `float`, `double`, `char`, and then outputting the values to the console.

Exercise 3-2. Modify the program in Exercise 3-1 to change the values of variables and output the values to the console a second time.

Exercise 3-3. Modify the program in Exercise 3-2 to declare a constant `float Pi` equal to 3.14159. Then assign a new value to pi (3.1) and output its value with the other variables. What happens when you try to compile this program?