Beginning C# 2008 Objects: From Concept to Code

Copyright © 2008 by Grant Palmer and Jacquie Barker

ISBN-13 (pbk): 978-1-4302-1088-7

ISBN-13 (electronic): 978-1-4302-1087-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Joohn Choe, Dominic Shakeshaft
Technical Reviewer: Andy Olsen
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell,
Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper,
Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Nancy Sixsmith
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Diana Van Winkle
Proofreader: Linda Seifert
Indexer: John Collin
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

A Little Taste of C#

If the first part of this book is supposed to be about general object concepts, then why on earth are we starting out with an introductory chapter on C#?

- It's indeed true that objects are "language neutral," so what you'll learn conceptually about objects in Part One of this book, and about object modeling in Part Two, could apply equally well to any object-oriented programming language (OOPL).
- We've found that seeing a sprinkling of code examples helps to cement object concepts; but we *could* have simply used language-neutral *pseudocode*—a natural-language way of expressing computer logic without worrying about the syntax of a specific language such as C#—for all our code examples in Parts One and Two.

This brings us back to our initial question: *why are we diving into C# syntax so soon?* Our reason for doing so is that we want you to become comfortable with C# syntax from the start because our goal for this book is not only to teach you about objects and object modeling but also to ultimately show you how objects translate into C# code. So although we do indeed use a bit of pseudocode to hide some of the more complex logic of our code examples throughout Parts One and Two, we focus for the most part on real C# syntax. Just remember that the object concepts you'll learn in Parts One and Two are equally applicable to other OOPLs unless otherwise noted.

In this chapter, you'll learn about the following:

- The many strengths of the C# programming language
- Predefined C# types, operators on those types, and expressions formed with those types
- The anatomy of a simple C# program
- The C# block structured nature
- Various types of C# expressions
- Loops and other control flow structures
- Printing messages to the screen, primarily for use in testing code as it evolves
- Elements of C# programming style

If you're a proficient C, C++, or Java programmer, you'll find much of C# syntax to be very familiar and should be able to breeze through this chapter fairly quickly.

If you've already been exposed to C# language basics, please feel free to skip to Chapter 2.

Getting Hands-On with C#

You're probably eager to get started writing, compiling, and running C# programs. But we're purposely not going to get into the details of downloading and installing C# and the .NET Framework on your computer, the mechanics of compiling programs, or any of that just yet. Here is a roadmap of the way this book is organized:

- Part One of the book focuses on object concepts—in other words, the "what" of objects; we don't want you to be distracted from learning these basic concepts by the bits and bytes of getting the C# environment up and running on your machine.
- Part Two of the book focuses on object modeling—that is, the "how" of designing an application to make the best use of objects. We don't want you to be trying to program without an appropriate OO "blueprint" to work from.
- You'll then be ready for the "grand finale"—rendering the object model in C# code to produce a working Student Registration System (SRS) application—in Part Three.

We'll be showing C# code throughout the book, but in Parts One and Two it will mostly be code snippets and simple examples. We'll wait until Part Three, when you have a good grounding in OO programming concepts, before we really dive into developing a complete C# application.

Why C#?

We *could* walk you through building the SRS using any OOPL. Why might we want to use C#? Read on and you'll quickly see why!

Practice Makes Perfect

The designers of C# were able to draw upon the lessons learned from other OOPLs that preceded it. They borrowed the best features of C++, Java, Eiffel, and Smalltalk, and then added some capabilities and features not found in those languages. Conversely, the features that proved to be most troublesome in earlier languages were eliminated. As a result, C# is a powerful programming language that is also easy to learn.

This is not to say that C# is a perfect language—no language is!—but simply that it has made some significant improvements over many of the languages that have preceded it.

C# Is Part of an Integrated Application Development Framework

The C# language is integrated into Microsoft's *.NET Framework*—Microsoft's powerful, comprehensive platform for developing applications and managing their runtime environment. The .NET Framework primarily supports the C#, C++, J#, and Visual Basic programming languages, but also provides a functionality called *cross-language interoperability* that allows objects created in different programming languages to work with each other. A core element of the .NET Framework is the *common language runtime (CLR)* that is responsible for the runtime management of any .NET Framework program. The CLR takes care of loading, running, and providing support services for the .NET Framework program.

The .NET Framework provides a high degree of interoperability between the languages it supports—C#, C++, Visual Basic, and JScript—through a *Common Language Specification (CLS)* that defines a common set of types and behaviors that every .NET language is guaranteed to recognize. The CLS allows developers to seamlessly integrate C# code with code written in any of the other .NET languages.

The .NET Framework also contains a vast collection of libraries called the *.NET Framework Class Library (FCL)* that provides almost all the common functionality needed to develop applications on the Windows platform. You'll find that with the FCL a lot of programming work has already been done for you on topics ranging from file access to mathematical functions to database connectivity. The C# language and the .NET Framework provide one-stop shopping for all your programming needs.

You can find out more about the .NET Framework here: `http://msdn.microsoft.com/en-us/library/default.aspx`.

C# Is Object-Oriented from the Ground Up

Before newer OOPLs such as C# and Java arrived on the scene, one of the most widely used OOPLs was C++, which is actually an object-oriented extension of the non-OOPL C. As such, C++ provides a lot of "back doors" that make it very easy to write decidedly "un-OO" code. In fact, many proficient C programmers transitioned to C++ as a better C without properly learning how to design an object-oriented application, and hence wound up using C++ for the most part as a procedural (non-OO) language.

In contrast, C# was built from the ground up to be a purely OOPL. As we'll discuss in more detail in the chapters that follow, *everything* in C# is an object:

- Primitive value types, such as `int` and `double`, inherit from the `Object` class.
- All the graphical user interface (GUI) building blocks—windows, buttons, text input fields, scroll bars, lists, menus, and so on—are objects.
- All functions are attached to objects and are known as *methods*—there can be no free-floating functions as there are in C/C++.
- Even the entry point for a C# program (now called the `Main` method) no longer stands alone, but is instead bundled within a *class*, the reasons for which we'll explore in depth in chapters to come.

Because of this, C# lends itself particularly well to writing applications that uphold the object-oriented paradigm. Yet, as we pointed out in the introduction to this book, merely using such an object-oriented language doesn't *guarantee* that the applications you produce will be *true* to this paradigm! You must be knowledgeable in *both* (a) how to design an application from the ground up to make the best use of objects and (b) how to apply the language correctly, our two primary intents of this book.

C# Is Free

One last valuable feature of C# that we'll mention is that it's *free*! You can download the C# compiler and all other libraries and utilities you'll need from the Microsoft Developer Network (MSDN) web site at no cost. We go into the details of setting up C# on your machine in Appendix A.

C# Language Basics

For those readers who have never seen C# code before, the rest of this chapter will present an introduction to the basic syntax of the C# programming language. Keep in mind that this is only a taste of C#, enough to help you understand the coding examples in Parts One and Two of this book. We'll revisit C# in substantially more depth in Part Three (Chapters 13 through 16), in which we'll delve much more deeply into the language in building a fully functional SRS application.

■**Note** If you haven't taken the time to read the introduction to this book, now is a good time to do so! The SRS application requirements are introduced as a case study there.

A Reminder About Pseudocode vs. Real C# Code

As mentioned in the beginning of this chapter, we occasionally use little bits of pseudocode in our code examples throughout Parts One and Two of the book to hide irrelevant logic details. To make it clear when we're using pseudocode versus real code, we used *`italic`* versus `regular code font`:

This is real C# syntax:

```
for (int i = 0; i <= 10; i++) {
```

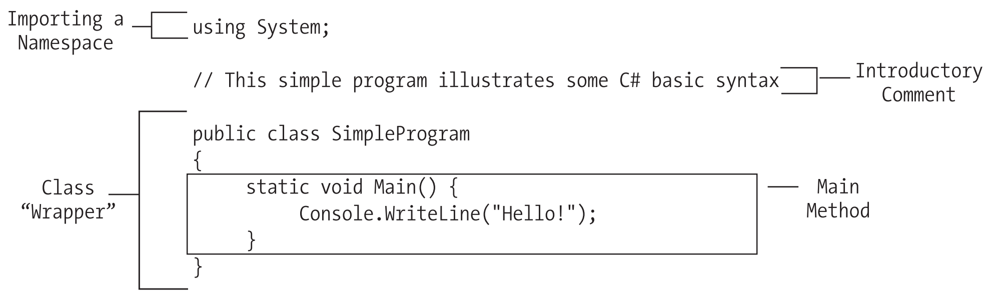
This is pseudocode:

```
    compute the grade for the ith Student
}
```

We'll remind you of this fact a few more times, so that you don't forget and accidentally try to type in and compile pseudocode somewhere along the way.

Anatomy of a Simple C# Program

One of the simplest of all C# applications, the classic "Hello" program, is shown in Figure 1-1.



```
using System;

// This simple program illustrates some C# basic syntax

public class SimpleProgram
{
     static void Main() {
          Console.WriteLine("Hello!");
     }
}
```

Figure 1-1. *Anatomy of a simple C# program*

Let's go over the key elements of our simple program.

The using System; Statement

The first line of the program is required for our program to compile and run properly, by providing the compiler with knowledge of the types in the `System` *namespace*, which is a logical grouping of predefined C# programming elements (in the case of C#, part of the FCL mentioned earlier):

```
using System;
```

We'll defer a detailed explanation of namespaces until Chapter 13; for now, simply realize that the `using System;` statement is required for this program to compile properly—specifically, for the line `Console.WriteLine("Hello!");` to compile properly.

`using` is a C# *keyword*. Keywords, also known as *reserved words*, are tokens that have special meaning in a language, so they cannot be used by programmers as the names of variables, functions, or any of the other C# building blocks that you'll be learning about. We'll encounter many more C# keywords throughout the book.

Comments

The next line of our program is a *single-line* comment:

```
// This simple program illustrates some basic C# syntax.
```

In addition to single-line comments, the C# language also supports the C language style of *delimited comments*, which can span multiple lines. Delimited comments begin with a forward slash followed by an asterisk (`/*`) and end with an asterisk followed by a forward slash (`*/`). Everything enclosed between these delimiters is treated as a comment and is therefore ignored by the compiler, no matter how many lines the comment spans:

```
/* This is a single line C-style comment. */

/* This is a multiline C-style comment. This is a handy way to temporarily
   comment out entire sections of code without having to delete them.
   From the time that the compiler encounters the first 'slash asterisk'
   above, it doesn't care what we type here; even legitimate lines of code,
   as shown below, are treated as comment lines and thus ignored by the
   compiler until the first 'asterisk slash' combination is encountered.
x = y + z;
a = b / c;
j = s + c + f;
*/
```

■ **Note** There is also a third type of C# comment that is used within XML document files. XML documentation comments are denoted by three slashes (///).

Note that comments can't be nested; that is, the following will *not* compile:

```
/* This starts a comment...
x = 3;

/* Whoops!  We are mistakenly trying to nest a SECOND comment
before terminating the FIRST!
This is going to cause us compilation problems, because the
compiler is going to IGNORE the start of this second/inner comment - we're IN
a comment already, after all! - and so as soon as we try to terminate
this SECOND/inner comment, the compiler will think that we've terminated the
FIRST/outer comment instead... */
z = 2;
 */ ...then, when we try to terminate the FIRST/outer comment,
            the compiler will inform us that THIS line of code is invalid.
```

When the compiler reaches what we intended to be the terminating `*/` of the "outer" comment in the last line of code, the following compiler error will be reported:

```
  error: Invalid expression term '/'
  error: ; expected
```

Class Declaration/"Wrapper"

Next comes a class "wrapper"—more properly termed a *class declaration*—of the following form where braces, `{...}`, enclose the main logic to be performed by the class, as well as enclosing other building blocks of a class:

```
class name

{...}

e.g.,

class SimpleProgram
{...}
```

In later chapters, you'll learn all about classes, how to name them, and in particular why you even need a class wrapper in the first place. For now, simply note that the token `class` is another one of C#'s keywords, whereas `SimpleProgram` is a name that we invented.

Main Method

Within the `SimpleProgram` class declaration, we find the starting function for the program, called the `Main` method in C#. The `Main` method serves as the entry point for a C# program. When the program executable is invoked, the system will call the `Main` method to launch our application.

■Note With trivial applications such as this simple example, all logic can be contained within this single method. For more complex applications, on the other hand, the `Main` method can't possibly contain all the logic for the entire system. You'll learn how to construct an application that transcends the boundaries of the `Main` method later in the book.

The first line of the method, shown here, defines what is known as the `Main` method's *header*, and must appear exactly as shown (with one minor exception that we'll explain in Chapter 13 having to do with optionally receiving arguments from the command line):

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

Our `Main` *method body*, enclosed in braces, `{...}`, consists of a single statement:

```
Console.WriteLine("Hello!");
```

This statement prints the following message to the screen:

```
Hello!
```

We'll talk more about this statement's syntax in a bit, but for now note the use of a semicolon at the end of the statement. As in C, C++, and Java, semicolons are placed at the end of individual C# statements. Braces `{...}` delimit *blocks* of code, the significance of which we'll discuss in more detail later in this chapter, in the section entitled "Code Blocks and Variable Scope."

Other things that we'd typically do inside of the `Main` method of a more elaborate program include declaring variables, creating objects, and calling other methods.

Now that we've looked at a simple C# program, let's explore some of the basic syntax features of C# in more detail.

Predefined Types

Generally speaking, C# is said to be a *strongly typed* programming language in that when a variable is declared, its type must also be declared. Among other things, declaring a variable's type tells the compiler how much memory to allocate for the variable.

The C# language and .NET Framework make use of the CTS, a specification that defines a set of types as well as the behavior of those types. The CTS defines a wide variety of types in two main families: *value types* and *reference types*. Both value types and reference types can also be declared to be *generic types*, which means that they can represent more than one type. In this chapter, we'll focus on C#'s *predefined value types*, also known as *simple types*, along with the `string` type, which happens to be a reference type.

The C# language supports a variety of simple types. The most commonly used types are as follows (all of them are C# keywords):

- `bool`: Boolean true or false value
- `char`: 16-bit Unicode character
- `byte`: 8-bit unsigned integer

- `short`: 16-bit signed integer
- `int`: 32-bit signed integer
- `long`: 64-bit signed integer
- `float`: 32-bit single-precision floating point
- `double`: 64-bit double-precision floating point

Each variable declared to be of a simple type represents a *single* integer, floating point, Boolean, byte, or character value.

Variables

As previously stated, before a variable can be used in a program, the type and name of the variable must be declared. An initial value can be supplied when a variable is first declared, or the variable can be assigned a value later in the program. For example, the following code snippet declares two simple type variables. The first variable, of type `int`, is given an initial value when the variable is declared. The second variable, of type `double`, is declared and then assigned a value on a subsequent line of code.

```
int count = 3;

double total;
// intervening code...details omitted
total = 34.3;
```

A value can be assigned to a `bool` variable using the `true` or `false` keywords:

```
bool blah;
blah = true;
```

Boolean variables are often used as flags to signal whether some code should be conditionally performed. An example follows:

```
bool error = false;  // Initialize the flag.
//...

// Later in the program (pseudocode):
if (some error situation arises) {
  // Set the flag to true to signal that an error has occurred.
  error = true;
}
//...

// Still later in the program:
if (error == true) {
     // Pseudocode.
     take corrective action
}
```

■**Note** We'll talk specifically about the syntax of the `if` statement, one of several different kinds of C# flow control statements, a bit later.

A literal value can be assigned to a variable of type `char` by surrounding the value (a single Unicode character) in *single* quotes as follows:

```
char c = 'A';
```

Variable Naming Conventions

Most variable names use what is known as *Camel casing*, wherein the first letter of the name is in lowercase, the first letter of each subsequent concatenated word in the variable name is in uppercase, and the rest of the characters are in lowercase.

■**Note** In subsequent chapters, we'll refine the rules for naming variables as we introduce additional object concepts.

For example, the following variable names follow the C# variable naming conventions:

```
int grade;
double averageGrade;
string myPetRat;
bool weAreFinished;
```

Recall that, as mentioned earlier, a C# keyword can't be used as a variable name:

```
int float;  // this won't compile—"float" is a keyword
```

Variable Initialization and Value Assignment

There are different types of variables in C# based on how and where the variables are declared in the program. Some variable types are *initialized* with a default value when they are declared. Local variables, those that are declared inside a method or other block of code, are given no default value when they are declared, so we must explicitly *assign* a value to a variable before the variable's value is accessed in a statement. For example, in the following code snippet, two local integer variables are declared: `foo` and `bar`. A value is assigned to the variable `foo`, but not to variable `bar`, and an attempt is made to add the two variables together:

```
static void Main() {
       int foo;    // local variable
       int bar;    // another local variable

       foo = 3;   // We're assigning a value to foo, but not bar.
       foo = foo + bar;   // this line won't compile
```

If we were to try to compile this code snippet, we would get the following compilation error message regarding the last line of the preceding code example:

```
error: use of unassigned local variable 'bar'
```

The compiler is telling us that the local variable `bar` has been declared, but its value is undefined. To correct this error, we need to assign an explicit value to `bar` before trying to add its value to `foo`:

```
int foo;
int bar;

foo = 3;
bar = 7;  // We're now assigning values to BOTH variables.

foo = foo + bar;  // This line will now compile properly.
```

■**Note** As it turns out, the story with respect to variable initialization is a bit more complex than what we've discussed here. You'll learn in Chapter 13 that the rules of automatic initialization are somewhat different when dealing with the inner workings of objects.

Strings

We'll discuss one more important predefined type in this chapter: the `string` type.

■**Note** Just a reminder: unlike the other C# types introduced in this chapter, `string` isn't a value type; it's a reference type, as we mentioned earlier. For purposes of this introductory discussion of strings, this observation isn't important; the significance of the `string` type's being a reference type will be discussed in Chapter 13.

A `string` represents a sequence of Unicode characters. There are several ways to create and initialize a `string` variable. The easiest and most commonly used way is to declare a variable of type `string` and to assign the variable a value using a *string literal*, which is any text enclosed in double quotes:

```
string name = "Zachary";
```

Note that we use double quotes, not single quotes, to surround a string literal when assigning it to a `string` variable, even if it consists of only a single character:

```
string shortString = "A";      // Use DOUBLE quotes when assigning a literal
                               // value to a string...
```

```
string longString = "supercalifragilisticexpialadocious";  // (ditto)

char c = 'A';                   //...and SINGLE quotes when assigning a
                                // literal value to a char.
```

Two commonly used approaches for assigning an initial value to a `string` variable as a placeholder are as follows:

- Setting it equal to an empty string, represented by two consecutive double quote marks:

  ```
  string s = "";
  ```

- Setting it equal to the reserved word `null`, which is the "zero equivalent" value for the `string` type (and, as you'll learn later on, for reference types/objects in general):

  ```
  string s = null;
  ```

The plus sign (+) operator is normally used for addition, but when used with `string` variables, it represents *string concatenation*. Any number of `string` variables or `string` literals can be concatenated together with the + operator:

```
string x = "foo";
string y = "bar";
string z = x + y + "!";  // z now equals "foobar!"; x and y are unchanged
```

You'll learn about some of the many other operations that can be performed with or on strings, along with gaining insights into their object-oriented nature, in Chapter 13.

Case Sensitivity

C# is a *case-sensitive* language. That is, the use of uppercase versus lowercase in C# is deliberate and mandatory. For example:

- Variable names that are spelled the same way but that differ in their use of case; e.g., `x` (lowercase) versus `X` (uppercase) represent *different* variables.
- All keywords are expressed in all lowercase: `public`, `class`, `int`, `bool`, and so forth. Don't get "creative" about capitalizing them because the compiler will complain!
- Capitalization of the name of the `Main` method is mandatory.

C# Expressions

A *simple expression* in C# is the following (plus a few more expression types having to do with objects that you'll learn about in Chapter 13):

- A constant: `7`, `false`
- A `char`(acter) literal: `'A'`, `'&'`
- A `string` literal: `"foo"`

- The name of any variable declared to be of one of the predefined types that we've discussed so far: `myString`, `x`
- Any *two* of the preceding items that are combined with one of the C# *binary operators* (discussed in detail later in this chapter): `x + 2`
- Any *one* of the preceding items that is modified by one of the C# *unary operators* (discussed in detail later in this chapter): `i++`
- Any of the preceding simple expressions enclosed in parentheses: `(x + 2)`

Assignment Statements

Assigning a value to a variable is accomplished by using the assignment operator `=`. An *assignment statement* consists of a (previously declared) variable name to the left of the `=`, and an expression that evaluates to the appropriate type to the right of the `=`; for example:

```
count = 1;

total = total + 4.0;  // assuming that total was declared to be a double variable

price = cost + (a + b)/length;  // assuming all variables properly declared
```

Arithmetic Operators

The C# language provides a number of basic arithmetic operators, as follows:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus (the remainder when the operand to the left of the `%` operator is divided by the operand to the right)

The + and – operators can also be used in prefix fashion to indicate positive or negative numbers: -3.7, +42.

In addition to the simple assignment operator, `=`, there are a number of specialized *compound assignment operators*, which combine variable assignments with an operation. The compound assignment operators for arithmetic operations are as follows:

`+=`	`a += b` is equivalent to `a = a + b`
`-=`	`a -= b` is equivalent to `a = a - b`
`*=`	`a *= b` is equivalent to `a = a * b`
`/=`	`a /= b` is equivalent to `a = a / b`
`%=`	`a %= b` is equivalent to `a = a % b`

Note The compound assignment operators don't add any new functionality; they are simply provided as a convenience to simplify code. For example, the statement

```
total = total + 4.0;
```

can be alternatively written as

```
total += 4.0;
```

The final two arithmetic operators that we'll introduce are the *increment* (++) and *decrement* (--) operators, which are used to increase or decrease the value of an integer variable by 1 or of a floating point value by 1.0. The increment and decrement operators can also be used on `char` variables. For example, consider the following code snippet:

```
char c = 'e';
c++;
```

When the code snippet is executed, the variable `c` will have the value `'f'`, which is the next character in the Unicode sorting sequence.

The increment and decrement operators can be used in either a *prefix* or *postfix* manner.

If the operator is placed *before* the variable it's operating on (*prefix* mode), the increment or decrement of that variable is performed *before* the variable's value is used in any assignments made via that statement.

If the operator is placed *after* the variable it's operating on (*postfix* mode), the increment or decrement occurs *after* the variable's value is used in any assignments made via that statement.

For example, consider the following code snippet, which uses the prefix increment (++) operator:

```
int a = 1;
int b = ++a;  // a will be incremented to 2, then b will be assigned the
              // value 2
```

After both lines of code have executed, the value of variable `a` will be 2 (as will the value of variable `b`) because in the second line of code, the increment of variable `a` (from 1 to 2) occurs *before* the value of `a` is assigned to variable `b`. The preceding two lines of code are logically equivalent to the following three lines:

```
int a = 1;
a = a + 1;
int b = a;
```

Now let's look at the same code snippet with the increment operator written in a postfix manner:

```
int a = 1;
int b = a++;  // b will be assigned the value 1, then a will be incremented
              // to 2
```

After both lines of code have executed, the value of variable b will be 1, whereas the value of variable a will be 2 because in the second line of code, the increment of variable a (from 1 to 2) occurs *after* the (old) value of a is assigned to variable b. The preceding two lines of code are logically equivalent to the following three lines:

```
int a = 1;
int b = a;
a = a + 1;
```

Here is a slightly more complex example:

```
int y = 1;
int z = 2;
int x = y++ * ++z;  // x will be assigned the value 3, because z will be
                    // incremented from 2 to 3 before its value is used in
                    // the multiplication, whereas y will remain at 1 until
                    // AFTER its value is used.
```

As you'll see in a bit, the increment and decrement operators are commonly used in loops and other flow of control structures.

Evaluating Expressions and Operator Precedence

Expressions of arbitrary complexity can be built up around the various different simple expression types by nesting parentheses; for example, ((((4/x) + y) * 7) + z). The compiler evaluates such expressions from innermost to outermost parentheses, left to right. Assuming that x, y, and z are declared and initialized as shown here:

```
int x = 1;
int y = 2;
int z = 3;
```

then the expression on the right side of the following assignment statement

```
int answer = ((8 * (y + z)) + y) * x;
```

would be evaluated piece by piece as follows:

$$\begin{gathered}((8 * \underline{(y + z)}) + y) * x \\ (\underline{(8 * 5)} + y) * x \\ \underline{(40 + y)} * x \\ \underline{42 * x} \\ 42\end{gathered}$$

In the absence of parentheses, certain operators take precedence over others in terms of when they will be applied in evaluating an expression. For example, multiplication or division is by default performed before addition or subtraction. The automatic precedence of one operator over another can be explicitly altered through the use of parentheses; operations inside parentheses will be performed before operations outside of them. Consider the following code snippet:

```
int j = 2 + 3 * 4;  // j will be assigned the value 14
int k = (2 + 3) * 4;   // k will be assigned the value 20
```

In the first line of code, which uses no parentheses, the multiplication operation takes precedence over the addition operation, so the overall expression evaluates to the value 2 + 12 = 14; it's as if we've explicitly written "2 + (3 * 4)" without having to do so.

In the second line of code, parentheses are explicitly placed around the operation 2 + 3 so that the addition operation will be performed first, and the resultant sum will then be multiplied by 4 for an overall expression value of 5 * 4 = 20.

Logical Operators

A *logical expression* compares two (simple or complex) expressions *exp1* and *exp2*, in a specified way, and evaluates to a Boolean value of `true` or `false`.

To create logical expressions, C# provides the following *relational operators*:

exp1 == *exp2*	`true` if *exp1* equals *exp2* (note use of a double equal sign)
exp1 > *exp2*	`true` if *exp1* is greater than *exp2*
exp1 >= *exp2*	`true` if *exp1* is greater or equal to *exp2*
exp1 < *exp2*	`true` if *exp1* is less than *exp2*
exp1 <= *exp2*	`true` if *exp1* is less than or equal to *exp2*
exp1 != *exp2*	`true` if *exp1* isn't equal to *exp2* (`!` is read as "not")
`!`*exp*	`true` if *exp* is false, and `false` if *exp* is true

In addition to the relational operators, C# provides *logical operators* that can be used in combination with the relational operators to create complex logical expressions that involve more than one comparison.

`&&`	Logical "and"
`||`	Logical "or"
`!`	Logical "not" (the `!` operator toggles the value of a logical expression from `true` to `false` and vice versa)

The logical "and" and "or" operators are binary operators; their left and right operands must both be valid logical expressions so that they evaluate to Boolean values. If the `&&` operator is used, both the left and right operands must be true for the compound logical expression to be true. With the `||` operator, the compound logical expression will be true if either the left or right operand is true.

Here is an example that uses the logical "and" operator to program the compound logical expression "if x is greater than 2.0 and y isn't equal to 4.0":

```
if (x > 2.0 && y != 4.0) {
  // Pseudocode.
  do some stuff...
}
```

Note that because the > and != operators take precedence over the && operator, we don't need to insert extra parentheses as shown here:

```
if ((x > 2.0) && (y != 4.0)) {
  // Pseudocode.
  do some stuff...
}
```

Logical expressions are most commonly seen in flow of control structures, discussed later in this chapter.

Implicit Type Conversions and Explicit Casting

C# supports *implicit type conversions*. For example, we try to assign the value of some variable y to another variable x, as shown here:

```
x = y;
```

The two variables were originally declared to be of different types; then C# will attempt to perform the assignment, automatically converting the type of the value of y to the type of x, but only if precision won't be lost in doing so. (C# differs from C and C++ in this regard because the latter two perform automatic type conversions even if precision is lost.) This is best understood by looking at an example:

```
int x;
double y;
y = 2.7;
x = y;  // Trying to assign a double value to an int variable; this line
        // will compile in C and C++, but not in C#.
```

In the preceding code snippet, we're attempting to copy the double value of y, 2.7, into x, which is declared to be an int. If this assignment were to take place, the fractional part of y would be truncated, and x would wind up with an integer value of 2. This represents a loss in precision, also known as a *narrowing conversion*. A C or C++ compiler will permit this assignment, thereby truncating the value; instead of assuming that this is what we intended to do, however, the C# compiler will generate an error on the last line:

```
Error::  Cannot implicitly convert type 'double' to type 'int'
```

To signal to the C# compiler that we're willing to accept the loss of precision, we must perform an *explicit cast*, which involves preceding the expression whose value is to be converted with the desired target type enclosed in parentheses. In other words, we'd have to rewrite the last line of the preceding example as follows for the C# compiler to accept it:

```
int x;
double y;
y = 2.7;
x = (int) y;   // This will compile now. The C# compiler 'relaxes',
               // because we have explicitly told it that we WANT the
               // narrowing conversion to occur.
```

Of course, if we were to reverse the direction of the assignment, as follows, the C# compiler would have no problem with the last statement, because in this particular case, we're assigning a value of less precision—2—to a variable capable of more precision; y will wind up with the value of 2.0:

```
int x;
double y;
x = 2;
y = x;           // Assign an int value to a double variable; y will assume
                 // the value 2.0.
```

This is known as a *widening conversion*; such conversions are performed automatically in C#, and need not be explicitly cast.

Note that there is an idiosyncrasy with regard to assigning constant values to a `float` in C#; the following statement will generate a compiler error because a numeric constant value with a fractional component such as 3.5 is automatically treated by C# as a more precise `double` value, so the compiler will once again refuse to make a transfer that causes precision to be lost:

```
  float y = 3.5;    // won't compile!
```

To make such an assignment, we must explicitly cast the floating point constant into a `float`:

```
  float y = (float)3.5;   // OK; we're using a cast here.
```

Alternatively, few can force the constant on the right side of the assignment statement to be treated as a float by using the suffix F, as shown here:

```
  float y = 3.5F;   // OK, because we're indicating that the constant is to be
        // treated as a float, not as a double.
```

■Tip Yet another option is to simply use `double` instead of `float` variables to represent floating point numeric values. We'll typically use a `double` instead of a `float` whenever we need to declare floating point variables in our SRS application, just to avoid these hassles of type conversion.

There are no implicit conversions to the `char` type, and the `bool` type can't be cast, either implicitly or explicitly, into another type.

You'll see other applications of casting that involve objects later in the book.

Loops and Other Flow of Control Structures

Very rarely will a program execute sequentially, line by line, from start to finish. Instead, the execution flow of the program will be conditional. It might be necessary to have the program execute a certain block of code if a condition is met or another block of code if the condition isn't met. A program might have to repeatedly execute the same block of code. The C# language provides a number of different types of loops and other flow of control structures to take care of these situations.

if Statements

The `if` statement is a basic conditional branch statement that executes one or more lines of code if a condition, represented as a logical expression, is satisfied. Alternatively, one or more lines of code can be executed if the condition is *not* satisfied by placing that code after the keyword `else`. The use of an `else` clause within an `if` statement is optional.

The basic syntax of the `if` statement is as follows:

```
if (condition) {
  execute whatever code is contained within the braces if condition is met
}
```

Or add an optional `else` clause:

```
if (condition) {
  execute whatever code is contained within the braces if condition is met
}
else {
  execute whatever code is contained within the braces if condition is NOT met
}
```

If only one executable statement follows either the `if` or (optional) `else` keyword, the braces can be omitted as shown here, but it's generally considered good practice to always use braces:

```
// Pseudocode.
if (condition)
   single statement to execute if true;
else

   single statement to execute if false;
```

A single Boolean variable as a simple form of Boolean expression can, of course, serve as the logical expression/condition of an `if` statement. For example, it's perfectly acceptable to write the following:

```
// Use this bool variable as a 'flag' that gets set to true when
// some particular operation is completed.
bool finished;

// Initialize it to false.
finished = false;

// Intervening code, in which the flag may get set to true...details omitted.

// Test the flag. This next line is equivalent to: if (finished == true) {
if (finished) {
    Console.WriteLine("we are finished");
}
```

In this case, the logical expression serving as the condition for the `if` statement corresponds to “if `finished`” or “if `finished` equals `true`”.

The `!` operator can be used to negate a logical expression, so that the block of code associated with an `if` statement is executed when the expression is `false`:

```
// equivalent to: if (finished == false)
if (!finished) {
    // If the finished variable is set to false, this code will execute.
    Console.WriteLine("we are not finished");
}
```

In this case, the logical expression serving as the condition for the `if` statement corresponds to “if *not* `finished`” or “if `finished` equals `false`”.

When testing for equality, remember that we must use two consecutive equal signs, not just one:

```
// Note use of double equal signs (==) to test for equality.
if (x == 3) {
  y = x;
}
```

■ **Note** A common mistake made by beginning C# programmers is to try to use a *single* equal sign to test for equality as in this example:

```
  if (x = 3) {...}
```

In C#, an `if` test must be based on a valid logical expression; `x = 3` isn’t a *logical* expression; it’s an *assignment* expression.

Although the preceding `if` statement doesn’t even compile in C#, it does compile in the C and C++ programming languages because in those languages, `if` tests are based on evaluating expressions to either the integer value 0 (equivalent to false) or nonzero (equivalent to true).

It’s possible to nest `if-else` constructs to test more than one condition. If nested, an inner `if` (plus optional `else`) statement is placed within the `else` part of an outer `if`.

A basic syntax for a two-level nested `if-else` construct is shown here:

```
if (condition1) {
  // execute this code
}
else {
  if (condition2) {
    // execute this alternate code
  }
  else {
    // execute this code if none of the conditions are met
  }
}
```

There is no limit to how many nested `if-else` constructs can be used, but try not to go too crazy with nesting.

The nested `if` statement shown in the preceding example may alternatively be written without using nesting as follows:

```
if (condition1) {
  // execute this code
}
else if (condition2) {
    // execute this alternate code
}
else {
    // execute this code if none of the conditions are met
}
```

The two forms are logically equivalent.

Here is an example that uses a nested `if-else` construct to determine the size of an employee's bonus based on the employee's sales and length of service:

```
using System;

public class IfDemo
{
  static void Main() {
    double sales = 40000.0;
    int lengthOfService = 12;
    double bonus;

    if (sales > 30000.0 && lengthOfService >= 10) {
      bonus = 2000.0;
    }
    else if (sales > 20000.0) {
      bonus = 1000.0;
    }
    else {
      bonus = 0.0;
    }

    Console.WriteLine("Bonus = " + bonus);
  }
}
```

Here's the output generated by this example code:

```
Bonus = 2000.0
```

switch Statements

A `switch` statement is similar to an `if-else` construct in that it allows the conditional execution of one or more lines of code. However, instead of evaluating a logical expression as an `if-else` construct does, a `switch` statement compares the value of an integer, char, enum, or `string` expression against values defined by one or more `case` labels. If a match is found, the code following the matching `case` label is executed. An optional `default` label can be included to define code that is to be executed if the integer, char, enum, or string expression matches none of the `case` labels.

The general syntax of a `switch` statement is as follows:

```
switch (expression) {
    case value1:
      // code to execute if expression matches value1
      break;
    case value2:
      // code to execute if expression matches value2
      break;
   // more case labels, as needed...
    case valueN:
      // code to execute if expression matches valueN
      break;
    default:
      // default code if no case matches
      break;
}
```

For example:

```
int x;

// x is assigned a value somewhere along the line...details omitted.

switch (x) {
    case 1:
      // Pseudocode.
      do something based on the fact that x equals 1
      break;
    case 2:
      // Pseudocode.
      do something based on the fact that x equals 2
      break;
    default:
      // Pseudocode.
      do something if x equals something other than 1 or 2
      break;
}
```

Note the following:

- The expression in parentheses following the `switch` keyword must be an expression that evaluates to a `string` or integer value.
- The values following the `case` labels must be constant values (a "hard-wired" integer constant, character literal, or a string literal).
- Colons, not semicolons, terminate the `case` and `default` labels.
- The statements following a given `case` label do not have to be enclosed in braces. They constitute a *statement list* rather instead of a code block.

Unlike an `if` statement, a `switch` statement isn't automatically terminated when a match is found and the code following the matching `case` label is executed. To exit a `switch` statement, a *jump statement* must be used—typically, a `break` statement. If a jump statement isn't included following a given `case` label, the execution will "fall through" to the next `case` or `default` label. This behavior can be used to our advantage: when the same logic is to be executed for more than one `case` label, two or more `case` labels can be stacked up back to back as shown here:

```
// x is assumed to have been previously declared as an int
switch (x) {
    case 1:
    case 2:
    case 3:
        // code to be executed if x equals 1, 2, or 3
        break;
    case 4:
        // code to be executed if x equals 4
        break;
}
```

A `switch` statement is useful for making a selection between a series of mutually exclusive choices. In the following example, a `switch` statement is used to assign a value to a variable named `capital` based on the value of a variable named `country`. If a match isn't found, the `capital` variable is assigned the value "not in the database".

```
using System;

public class SwitchDemo
{
  static void Main() {
    string country;
    string capital;
    country = "India";

    // This switch statement compares the value of the variable "country"
    // against the value of three case labels. If no match is found,
    // the code after the default label is executed.
    switch (country) {
      case "England":
        capital = "London";
        break;
      case "India":
        capital = "New Delhi";
        break;
      case "USA":
        capital = "Washington D.C.";
        break;
      default:
        capital = "not in the database";
        break;
    }

    Console.WriteLine("The capital of " + country + " is " + capital);
  }
}
```

Here's the output for the preceding code example:

```
The capital of India is New Delhi
```

for Statements

A `for` statement is a programming construct that is used to execute one or more statements a certain number of times. The general syntax of the `for` statement is as follows:

```
for (initializer; condition; iterator) {
  // code to execute while condition is true
}
```

A `for` statement defines three elements that are separated by semicolons and placed in parentheses after the `for` keyword.

The *initializer* is typically used to provide an initial value for a *loop control variable*. The variable can be declared as part of the initializer or it can be declared earlier in the code, ahead of the `for` statement. For example:

```
  // The loop control variable 'i' is declared within the for statement:
     for (int i = 0; condition; iterator) {
      // code to execute while condition is true
  }
     // Note that i goes out of scope when the 'for' loop exits.
```

or:

```
  // The loop control variable 'i' is declared earlier in the program:
     int i;

  for (i = 0; condition; iterator) {
      // code to execute while condition is true
  }
  // Note that because i is declared before the for loop begins in this
  // case, i remains in scope when the 'for' loop exits.
```

The *condition* is a logical expression that typically involves the loop control variable:

```
for (int i = 0; i < 5; iterator) {
    // code to execute as long as i is less than 5
}
```

The *iterator* typically increments or decrements the loop control variable:

```
for (int i = 0; i < 5; i++) {
    // code to execute as long as i is less than 5
}
```

Again, note the use of a semicolon (;) after the initializer and condition, but *not* after the iterator.

Here's a breakdown of how a for loop operates:

- When program execution reaches a for statement, the initializer is executed first (and only once).
- The condition is then evaluated. If the condition evaluates to true, the block of code following the parentheses is executed.
- After the block of code finishes, the iterator is executed.
- The condition is then reevaluated. If the condition is still true, the block of code and update statement are executed again.

This process repeats until the condition becomes false, at which point the for loop exits.

Here is a simple example of using nested for statements to generate a simple multiplication table. The loop control variables j and k are declared inside their respective for statements. As long as the conditions in the respective for statements are met, the block of code following the for statement is executed. The ++ operator is used to increment the values of j and k after each time the respective block of code is executed.

```
using System;

public class ForDemo
{
  static void Main() {
    // Compute a simple multiplication table.

    for (int j = 1; j <= 4; j++) {
      for (int k = 1; k <= 4; k++) {
        Console.WriteLine(j + " * " + k + " = " + (j * k));
      }
    }
  }
}
```

Here's the output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

■**Note** Note the use of the `string` concatenation operator + in the `ForDemo` example; `string` representations of the value of `int` variables `j` and `k` are concatenated to the `string` literals `" * "`, and `" = "`.

Each of the three elements inside the parentheses of a `for` statement is optional (although the two separating semicolons are mandatory).

If the initializer is omitted, the loop control variable must have been declared and initialized before the `for` statement is encountered:

```
int i = 0;
for (; i < 5; i++) {
    // do some stuff as long as i is less than 5
}
```

If the iterator is omitted, we must make sure to take care of explicitly updating the loop control variable within the body of the `for` loop to avoid an infinite loop:

```
for (int i = 0; i < 5; ) {
    // do some stuff as long as i is less than 5

  // Explicitly increment i.
  i++;
}
```

If the condition is omitted, it will always be evaluated as being true, and the result is a potentially infinite loop:

```
for (;;) {
    // infinite loop!
}
```

■**Note** In the section titled "Jump Statements" later in this chapter, you'll see that jump statements can be used to break out of a loop.

As with other flow of control structures, if only one statement is specified after the `for` condition, the braces can be omitted (but it is considered good programming practice to use braces regardless):

```
for (int i = 0; i < 3; i++)
   sum = sum + i;
```

while Statements

A `while` statement is similar in function to a `for` statement, in that both are used to repeatedly execute an associated block of code. However, if the number of times that the code is to be executed is unknown when the loop first begins, a `while` statement is the preferred choice because a `while` statement continues to execute as long as a specified condition is met.

The general syntax for the `while` statement is as follows:

```
while (condition) {
  // code to execute while condition is true
}
```

The condition can be either a simple or complex logical expression that evaluates to a `true` or `false` value; for example:

```
int x = 1;
int y = 1;

while (x < 20 || y < 10) {
  // Pseudocode.
  presumably do something that affects the value of either x or y
}
```

When program execution reaches a `while` statement, the condition is evaluated first. If `true`, the block of code following the condition is executed. When the block of code is finished, the condition is evaluated again; if it is still `true`, the process repeats itself until the condition evaluates to `false`, at which point the `while` loop exits.

Here is a simple example illustrating the use of a `while` loop. A `bool` variable named `finished` is initially set to `false`. The `finished` variable is used as a flag: as long as `finished` is `false`, the block of code following the `while` loop will continue to execute. Presumably, there will be a statement inside the block of code that will eventually set `finished` to `true`, at which point the `while` loop will exit the next time the condition is retested.

```
using System;

public class WhileDemo
{
  static void Main() {
    bool finished = false;
    int i = 0;
    while (!finished) {
      Console.WriteLine(i);
      i++;
      if (i == 3)
        finished = true; // toggle the flag value
    }
  }
}
```

Here's the output:

```
0
1
2
```

As with the other flow of control structures, if only one statement is specified after the condition, the braces can be omitted (but it is considered good programming practice to use braces regardless):

```
while (x < 20)
   x = x * 2;
```

do Statements

With a `while` loop, the condition is evaluated before the code block following it is (conditionally) executed. Thus, it is possible that the code inside the loop body will never be run if the condition is false from the start. A `do` loop is similar to a `while` loop, except that the block of code is executed *before* the condition is evaluated. Therefore, we are guaranteed that the code block of the loop will be executed at least once.

The general syntax of a `do` statement is as follows:

```
do {
  // code to execute
} while (condition);
```

As was the case with the `while` statement, the condition of a `do` statement is a logical expression that evaluates to a Boolean value. A semicolon is placed after the parentheses surrounding the condition to signal the end of the `do` statement. We typically use a `do` loop when we know that we need to perform at least one iteration of a loop for initialization purposes.

```
    bool flag;

    do {
      // perform some code regardless of initial setting of 'flag', then
      // evaluate whether the loop should iterate again based on the value
      // of 'flag'.  The value of 'flag' can be set to true or false within
      // the loop to indicate whether the loop should execute again.
    } while (flag);
```

Jump Statements

Some of the loops and flow of control structures we have discussed will exit automatically when a condition is met (or not met), and some of them will not. The C# language defines a number of jump statements that are used to redirect program execution to another statement elsewhere in the code. The two types of jump statements that we will discuss in this section are the `break` and `continue` statements. Another jump statement, the `return` statement, is used to exit a method. We will defer our discussion of the `return` statement until Chapter 4.

You have already seen `break` statements in action earlier in this chapter, when they were used in conjunction with a `switch` statement. A `break` statement can also be used to abruptly terminate a `do`, `for`, or `while` loop. When a `break` statement is encountered during loop execution, the loop immediately terminates, and program execution is transferred to the line of code immediately after the loop or flow of control structure.

```
// This loop is intended to execute four times...
for (int j = 1; j <= 4; j++) {
  //...but, as soon as j attains a value of 3, the following 'if'
  // test passes, the break statement that it controls executes, and we
  // 'break out of' the loop.
  if (j == 3)
     break;

  // If, on the other hand, the 'if' test fails, we skip over the
  // 'break' statement, print the value of j, and keep on looping
  Console.WriteLine(j);
}

// The break statement, if/when executed, takes us to this line of code
// immediately after the loop.
Console.WriteLine("Loop finished");
```

The output produced by the preceding code snippet would be as follows:

```
1
2
Loop finished
```

A `continue` statement, on the other hand, is used to exit from the current iteration of a loop without terminating overall loop execution. A `continue` statement transfers program execution back up to the top of the loop (to the iterator part of a `for` loop) without finishing the particular iteration that is already in progress.

```
// This loop is intended to execute four times...
for (int j = 1; j <= 4; j++) {
  //...but, as soon as j attains a value of 3, the following 'if' test
  // passes and we 'jump' back to the j++ part of the for statement,
  // with j being incremented to 4...
  if (j == 3)
     continue;
```

```
        //...and so the following line doesn't get executed when j equals 3,
        // but DOES get executed when j equals 1, 2, and 4.
        Console.WriteLine(j);
      }
      Console.WriteLine("Loop finished");
```

The output produced by this code would be as follows:

```
1
2
4
Loop finished
```

Excessive use of `break` and `continue` statements in loops can result in code that is difficult to follow and maintain, so it's best to use these statements only when you have to use them.

Code Blocks and Variable Scope

C# (like C, C++, and Java) is a *block structured language*. As mentioned earlier in the chapter, a "block" of code is a series of zero or more lines of code enclosed within braces, like so: `{...}`.

- A method declaration, like the `Main` method of our `SimpleProgram`, defines a block.
- A class declaration, like the `SimpleProgram` class as a whole, also defines a block.
- As you have seen, many *control flow statements* also involve defining blocks of code.

Blocks can be nested inside one another to any arbitrary depth:

```
public class SimpleProgram
{
  // We're inside of the 'class' block (one level deep).
  static void Main() {
    // We're inside of the 'Main method' block (two levels deep).
    int x = 3;
    int y = 4;
    int z = 5;

    if (x > 2) {
      // We're now one level deeper (level 3), in a nested block.
      if (y > 3) {
        // We're one level deeper still (level 4), in yet another
        // nested block.
        // (We could go on and on!)
      } // We've just ended the level 4 block.
      // (We could have additional code here, at level 3.)
    } // Level 3 is done!
    // (We could have additional code here, at level 2.)
  } // That's it for level 2!
  // (We could have additional code here, at level 1.)
} // Adios, amigos! Level 1 has just ended.
```

The *scope* of a variable name is defined as that portion of code in which a name remains defined to the compiler, typically from the point where it is first declared down to the closing (right) brace for the block of code that it was declared in. A variable is said to be *in scope* only inside the block of code in which it is declared. Once program execution exits a block of code, any variables that were declared inside that block go out of scope and will be inaccessible to the program.

As an example of the consequences of variable scope, let's write a program called ScopeDemo, shown next. The ScopeDemo class declares three nested code blocks: one for the ScopeDemo class declaration, one for the Main method, and one as part of an if statement inside the body of the Main method.

```
public class ScopeDemo
{
  static void Main() {
    double cost = 2.65;

    if (cost < 5.0) {
      double discount = 0.05;  // declare a variable inside the 'if' block
      // other details omitted...
    }

    // When the 'if' block exits, the variable 'discount' goes out of scope,
    // and is no longer recognized by the compiler. If we try to use it
    // in a subsequent statement, the compiler will generate an error.

    double refund = cost * discount;   // this won't compile - discount is
  }                                    // no longer in scope
}
```

In the preceding example, a variable named cost is declared inside the block of code comprising the Main method body. Another variable named discount is declared inside the block of code associated with the if statement. When the if statement block of code exits, the discount variable goes out of scope. If we try to access it later in the program, as we do in the following line of code:

```
double refund = cost*discount;
```

The compiler will generate the following error:

```
error:: The name 'discount' does not exist in the current context
```

Note that a variable declared in an *outer* code block *is* accessible to any *inner* code blocks that follow the declaration. For example, in the preceding ScopeDemo example, the variable cost is accessible inside the nested if statement code block that follows its declaration.

Printing to the Screen

Most applications communicate information to users by displaying messages via the application's GUI. However, it is also useful at times to be able to display simple text messages to the command-line window from which we are running a program as a "quick and dirty" way of verifying that a program is working properly (you'll learn how to run C# programs from the command line in Chapter 13). Until we discuss how to craft a C# GUI in Chapter 16, this will be our program's primary way of communicating with the "outside world."

To print text messages to the screen, we use the following syntax:

```
Console.WriteLine(expression to be printed);
```

The `Console.WriteLine` method can accept very complex expressions and does its best to ultimately turn them into a single `string` value, which then gets displayed on the screen. Here are a few examples:

```
Console.WriteLine("Hi!");   // Printing a string literal/constant.

string greeting = "Hi!";
Console.WriteLine(greeting);   // Printing the value of a string variable.

string s = "foo";
string t = "bar";
Console.WriteLine(s + t);   // Using the string concatenation operator (+)
                            // to print "foobar".

int x = 3;
int y = 4;

Console.WriteLine(x);      // Converts x's int value into a string and
                           // prints the value "3" to the screen.

Console.WriteLine(x + y);  // Computes the sum of x and y, then
                           // prints the value "7" to the screen.
```

Note in the last line of code that the plus sign (+) is interpreted as the *integer* addition operator, not as the `string` concatenation operator, because it separates two variables that are both declared to be of type `int`. So, the sum of 3 + 4 is computed to be 7, which is then printed. In the next example, however, we get different (and arguably undesired) behavior:

```
Console.WriteLine("The sum of x plus y is:  " + x + y);
```

The preceding line of code causes the following to be printed:

```
The sum of x plus y is:  34
```

Why is this?

We evaluate most expressions from left to right, so because the first of the two plus signs separates a `string` literal and an `int`, it is interpreted as a `string` concatenation operator, and the value of `x` is thus converted into a `string`, producing the intermediate `string` value `"The sum of x plus y is: 3"`.

The second plus sign separates this intermediate `string` value from an `int` as well (`y`), so it is also interpreted as a `string` concatenation operator, and the value of `y` is thus converted into a `string`, producing the final `string` value `"The sum of x plus y is: 34"`, which is what finally gets printed.

To print the correct sum of x and y, we must force the second plus sign to be interpreted as an integer addition operator by enclosing the addition expression in nested parentheses:

```
Console.WriteLine("The sum of x plus y is: " + (x + y));
```

The nested parentheses cause the innermost expression to be evaluated first; the second plus sign is now seen by the compiler as separating two `int` values, and will thus serve as the integer addition operator. Then the first plus sign is seen as separating a `string` from an `int`, and is thus treated as a string concatenation operator, ultimately causing this print statement to display the correct message on the screen:

```
The sum of x plus y is: 7
```

When writing code that involves complex expressions, it is a good idea to use parentheses liberally to make our intentions clear to the compiler. Extra parentheses never hurt!

Write versus WriteLine

When we use `Console.WriteLine(...)`, whatever expression is enclosed inside the parentheses will be printed, followed by a *line terminator*. The following code snippet:

```
Console.WriteLine("First line.");
Console.WriteLine("Second line.");
Console.WriteLine("Third line.");
```

produces this as output:

```
First line.
Second line.
Third line.
```

By contrast, the following statement causes whatever expression is enclosed in parentheses to be printed *without* a line terminator:

```
Console.Write(expression to be printed);
```

Using `Write` in combination with `WriteLine` allows us to build up a single line of output with a series of `Write` statements, as shown by the following example:

```
Console.Write("C");       // Using Write here.
Console.Write("SHA");     // Using Write here.
Console.WriteLine("RP");  // Note use of WriteLine as the last statement.
```

This code snippet produces the single line of output:

```
CSHARP
```

We can make a program listing more readable by breaking up the contents of a long print statement into multiple concatenated `strings`, and then breaking the statement along plus-sign boundaries:

```
    statement;
    another statement;
    Console.WriteLine("Here is an example of how " +
                      "to break up a long print statement " +
                      "with plus signs.");

    yet another statement;
```

Even though the preceding statement is broken across three lines of code, it will be printed as a single line of output:

```
Here is an example of how to break up a long print statement with plus signs.
```

Note that without the + signs in the preceding example, the code would not compile because a string literal won't automatically carry over onto the next line.

Escape Sequences

C# defines a number of *escape sequences* so that we can represent special characters, such as newline and tab characters, in `string` expressions. The most commonly used escape sequences are listed here:

`\n`	Newline
`\b`	Backspace
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote

One or more escape sequences can be included in the expression that is passed to the `Write` and `WriteLine` methods. For example, consider the following code snippet:

```
    Console.WriteLine("Presenting...");
    Console.WriteLine("\n...for a limited \"time\" only...\n");
    Console.WriteLine("\tBailey the Wonder Dog!");
```

When the preceding code is executed, the following output is displayed:

```
Presenting...

...for a limited "time" only...

        Bailey the Wonder Dog!
```

There is a blank line before and after the second line of output because we inserted extra `\n` escape sequences in the second statement, the word `"time"` is quoted because of our use of the `\"` escape sequences in that same statement, and the third line of output has been tabbed over one position to the right by virtue of our use of `\t`.

Elements of C# Style

One of the trademarks of good programmers is that they produce readable code. Your professional life will probably not involve generating code by yourself on a mountaintop, so your colleagues will need to be able to work with and modify your programs. Here are some guidelines and conventions that will help you to produce clear, readable C# programs.

Proper Use of Indentation

One of the best ways to make C# programs readable is through proper use of indentation to clearly delineate statement hierarchies. Statements within a block of code should be indented relative to the starting/end line of the enclosing block (that is, indented relative to the lines carrying the braces). The examples in the MSDN web pages use four spaces, but some programmers use two spaces and others prefer three. The examples in this book use a two-space indentation convention.

■**Tip** If you are using Visual Studio, you can configure the indentation spacing as one of the VS options.

To see how indentation can make a program readable, consider the following two programs. In the first program, no indentation is used:

```
using System;

public class StyleDemo
{
static void Main() {
string name = "cheryl";
for (int i = 0; i < 4; i++) {
if (i != 2) {
Console.WriteLine(name + " " + i);
}
}
Console.WriteLine("what's next");
}
}
```

It is easy to see how someone would have to go through this program very carefully to figure out what it is trying to accomplish; it's not very readable code.

Now let's look at the same program when proper indentation is applied. Each statement within a block is indented two spaces relative to its enclosing block. It's much easier now to see what the code is doing. It is clear, for example, that the `if` statement is inside of the `for` statement's code block. If the `if` statement condition is true, the `WriteLine` method is called. It is also obvious that the last `WriteLine` method call is outside of the `for` loop. Both versions of this program produce the same result when executed, but the second version is much more readable.

```
using System;

public class StyleDemo
{
  static void Main() {
    string name = "Cheryl";
    for (int i = 0; i < 4; i++) {
      if (i != 2) {
        Console.WriteLine(name + " " + i);
      }
    }
    Console.WriteLine("What's next");
  }
}
```

This code's output is shown here:

```
Cheryl 0
Cheryl 1
Cheryl 3
What's next
```

Failure to properly indent makes programs unreadable and hence harder to debug—if a compilation error arises because of imbalanced braces, for example, the error message often occurs much later in the program than where the problem exists. For example, the following program is missing an opening brace on line 11, but the compiler doesn't report an error until line 25!

```
using System;
public class Indent2
{
  static void Main() {
    int x = 2;
    int y = 3;
    int z = 1;

    if (x >= 0) {
      if (y > x) {
        if (y > 2) // missing opening brace here on line 11, but...
          Console.WriteLine("A");
          z = x + y;
        }
        else {
          Console.WriteLine("B");
          z = x - y;
        }
      }
      else {
        Console.WriteLine("C");
        z = y - x;
      }
    }
    else Console.WriteLine("D");  // compiler first complains here!(line 25)
  }
}
```

The error message that the compiler generates in such a situation is rather cryptic; it points to line 25 as the problem and doesn't really help us much in locating the real problem on line 11:

```
IndentDemo.cs (25,5) error:
Invalid token 'else' in class, struct, or interface member declaration.
```

However, at least we've properly indented, so it will likely be easier to hunt down the missing brace than it would be if our indentation were sloppy.

Sometimes we have so many levels of nested indentation, or individual statements are so long, that lines "wrap" when viewed in an editor or printed as hardcopy:

```
while (a < b) {
    while (c > d) {
      for (int j = 0; j < 29; j++) {
        x = y + z + a + b – 125*
(c * (d / e) + f) - g + h + j - l - m - n + o +
p * q / r + s;
      }
    }
}
```

To avoid this, it is best to break the line in question along white space or punctuation boundaries:

```
while (a < b) {
  while (c > d) {
    for (int j = 0; j < 29; j++) {
      // This is cosmetically preferred.
      x = y + z + a + b – 125*(c * (d / e) + f) - g +
          h + j - l - m - n + o + p * q / r + s;
    }
  }
}
```

Use Comments Wisely

Another important feature that makes code more readable is the liberal use of meaningful comments. Always keep in mind when writing code that you know what you are trying to do, but someone else trying to read your code may not. (We sometimes even need to remind *ourselves* of why we did what we did if we haven't looked at code that *we've* written in awhile!)

If there can be any doubt as to what a section of code does, add a comment:

- Include enough detail in the comment to clearly explain what you mean.
- Make sure that the comment adds value; don't state the obvious. The following is a fairly useless comment because it states the obvious:

  ```
  // Declare x as an integer, and assign it an initial value of 3.
  int x = 3;
  ```

- Indent each comment to the same level as the block of code or statement to which it applies.

For an example of how comments are important in making code readable, let's revisit an example from earlier in the chapter:

```
using System;

public class IfDemo
{
  static void Main() {
    double sales = 40000.0;
    int lengthOfService = 12;
    double bonus;

    if (sales > 30000.0 && lengthOfService >= 10) {
      bonus = 2000.0;
    }
    else {
      if (sales > 20000.0) {
        bonus = 1000.0;
      }
      else {
        bonus = 0.0;
      }
    }

    Console.WriteLine("Bonus = " + bonus);
  }
}
```

Because of the lack of comments, someone trying to read the code might have difficulty figuring out what business logic this program is trying to apply. Now let's look at the same program when clear, descriptive comments have been included in the code listing:

```
using System;

// This program computes the size of an employee's bonus.
//
// Written on March 5, 2008 by Jacquie Barker and Grant Palmer.
```

```
public class IfDemo
{
  static void Main() {
    // Quarterly sales in dollars.
    double sales = 40000.0;

    // Length of employment in months.
    int lengthOfService = 12;

    // Amount of bonus to be awarded in dollars.
    double bonus;

    // An employee gets a $2K bonus if (a) they've sold more than $30K this
    //quarter and (b) they've worked for the company for 10 months or more.
    if (sales > 30000.0 && lengthOfService >= 10) {
      bonus = 2000.0;
    }
    else {
      // Otherwise, ANY employee who has sold more than $20K this quarter
      // earns a bonus of $1K, regardless of how long they've worked for
      // the company.
      if (sales > 20000.0) {
        bonus = 1000.0;
      }
      // Employees who have sold less than $20K earn no bonus.
      else {
        bonus = 0.0;
      }
    }

    Console.WriteLine("Bonus = " + bonus);
  }
}
```

The program is now much more understandable because the comments explain what each section of the code is intended to accomplish.

Placement of Braces

For block structured languages that use braces, {...}, to delineate the start/end of blocks (for example, C, C++, Java, C#), there are two general schools of thought as to where the left/opening brace of a code block should be placed.

The first style is to place the left brace at the end of the line of code that starts the block and the matching right/closing brace on a line by itself:

```
 public class Test { //  Left brace on same line as class declaration

     static void Main() {  //  Ditto for method headers
```

```
        for (int i = 0; i < 3; i++) { //  And again for control flow blocks
          Console.WriteLine(i);

        //  Each closing brace goes on its own line:
        }
      }
    }
```

An alternative opening brace placement style is to place every opening brace on a line by itself:

```
public class Test
{
    static void Main()
    {
        for (int i = 0; i < 3; i++)
        {
          Console.WriteLine("i");
        }
      }
    }
```

Another possibility is a hybrid of these two approaches: the second style (brace on a separate line) is used for class declarations, and the first style (brace on the same line) is used for virtually everything else:

```
//  Left brace for class declaration on its own line:
public class Test
{
  //  but all others are on the same line as the initial line of code.
  static void Main() {

        for (int i = 0; i < 3; i++) {
          Console.WriteLine(i);

        //  Each closing brace goes on its own line:
        }
      }
    }
```

There is no absolute right or wrong style because the compiler doesn't care one way or the other. It is a good practice to maintain consistency in your code, however, so pick a brace placement style and stick with it.

Either way, it is important that the closing brace for a given block be indented the same number of spaces as the first line of code in the block so that they visually line up, as was discussed earlier.

Self-Documenting Variable Names

As with indentation and comments, the goal when choosing variable names is to make a program as readable, and hence self-documenting, as possible. Avoid using single letters as variable names, except for loop control variables. Abbreviations should be used sparingly, and only when the abbreviation is commonly used and widely understood by developers. Consider the following variable declaration:

```
int grd;
```

It's not completely clear what the variable name `grd` is supposed to represent. Is the variable supposed to represent a grid, a grade, or a gourd? A better practice is to spell the entire word out:

```
int grade;
```

At the other end of the spectrum, names that are too long, such as the following example, can make a code listing overwhelming to anyone trying to read it:

```
double averageThirdQuarterReturnOnInvestment;
```

It can sometimes be challenging to reduce the size of a variable name and still keep it descriptive, but do try to keep the length of your variable names within reason.

■**Note** We'll talk about naming conventions for other OO building blocks, such as methods and classes, as we introduce these topics later in the book.

The .NET Framework provides a series of naming guidelines to promote a uniform style across C# programs. If you want to read the complete details on the C# naming conventions, the guidelines can be found at the following site: `http://msdn.microsoft.com/en-us/library/ms229045.aspx`.

Summary

In this chapter, we discussed some of the advantages of C#, including the following:

- C# is an intuitive OOPL that improves upon many languages that preceded it.
- C# was designed from the ground up to be fully object-oriented.
- C# is part of (and thus has access to the power of) Microsoft's .NET Framework.
- C# can be downloaded for free from the MSDN web site.

■**Note** Of course, we haven't sung *all* of C#'s praises in this chapter; there are many other features that make C# a powerful OOPL that we'll cover in subsequent chapters.

In addition to exploring some of the advantages of C#, we also introduced you to some basic elements of C# syntax. In particular, we did the following in this chapter:

- Presented the anatomy of a simple C# program
- Discussed the predefined simple types and the `string` type
- Examined how value type variables are declared and initialized
- Introduced how a value of one type can be cast into a different type
- Discussed arithmetic, assignment, and logical expressions
- Presented loops and other flow of control structures available with C#
- Explored how to define blocks of code and the concept of variable scope
- Learned how to print text messages to the console with the `Write` and `WriteLine` methods
- Discussed some basic elements of good C# programming style

There's a lot more to learn about C#—things you'll need to know in building the SRS application in Part Three of the book—but we need to explain a number of basic object concepts first. So, on to Chapter 2!

Exercises

1. Research Microsoft's C# Language Tour web site at `http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx`.

   Cite any advantages or features of C# not mentioned in this chapter.

2. Explore the Microsoft .NET Framework home page at `http://msdn.microsoft.com/en-us/library/w0x726c2.aspx`.

   Remember that C# can make use of all the libraries and other capabilities provided by the .NET Framework.

3. Using a `for` loop and a `continue` statement, create a code snippet that will write the even numbers from 1 to 10 to the console.

4. Using what you know about defining blocks of code and proper indentation technique, make the following code snippet more readable:

```
int count = 0;
for (int j = 0; j < 2; j++) {
count = j;
for (int k = 0; k < 3; k++)
count++;
Console.WriteLine("count = " + count);
}
```

5. Compare what you've learned about C# so far to another programming language that you are already familiar with. What is similar about the two languages? What is different?

6. Given these initial variable declarations and value assignments:

   ```
   int a = 1;
   int b = 1;
   int c = 1;
   ```

   evaluate the following expression:

   ```
   ((((c++ + --a) * b) != 2) && true)
   ```