CHAPTER 1

A First Program Using C#

In this chapter you will:

- Learn about programming
- Learn about procedural and object-oriented programming
- Learn about the features of object-oriented programming languages
- Learn about the C# programming language
- Write a C# program that produces output
- Learn how to select identifiers to use within your programs
- Improve programs by adding comments and using the `System` namespace
- Compile and execute a C# program using the command prompt and using Visual Studio

Programming a computer is an interesting, challenging, fun, and sometimes frustrating task. As you learn a programming language, you must be precise and careful as well as creative. Computer programmers can choose from a variety of programming languages, such as Visual Basic, Java, and C++. C# (pronounced "C Sharp") is a newer programming language that offers a wide range of options and features. As you work through this book, you will master many of them, one step at a time. If this is your first programming experience, you will learn new ways to approach and solve problems and to think logically. If you know how to program but are new to C#, you will be impressed by its capabilities.

In this chapter, you will learn about the background of programming that led to the development of C#, and you will write and execute your first C# programs.

Programming

A computer **program** is a set of instructions that tells a computer what to do. Programs are also called **software;** software comes in two broad categories:

- **System software** describes the programs that operate the computer. Examples include operating systems such as Microsoft Windows, Mac OSX, and Linux.
- **Application software** describes the programs that allow users to complete tasks such as creating documents, calculating paychecks, and playing games.

The physical devices that make up a computer system are called **hardware**. Internally, computer hardware is constructed from circuitry that consists of small on/off switches; the most basic circuitry-level language that computers use to control the operation of those switches is called **machine language**. Machine language is expressed as a series of *1*s and *0*s—*1*s represent switches that are on, and *0*s represent switches that are off. If programmers had to write computer programs using machine language, they would have to keep track of the hundreds of thousands of *1*s and *0*s involved in programming any worthwhile task. Not only would writing a program be a time-consuming and difficult task, but modifying programs, understanding others' programs, and locating errors within programs all would be cumbersome. Additionally, the number and location of switches vary from computer to computer, which means you would need to customize a machine-language program for every type of machine on which the program had to run.

Fortunately, programming has become easier because of the development of high-level programming languages. A **high-level programming language** allows you to use a limited vocabulary of reasonable keywords. **Keywords** are predefined and reserved identifiers that have special meaning in a language. In other words, high-level language programs contain words such as *read*, *write*, or *add* instead of the sequence of on/off switches that perform these tasks. High-level languages also allow you to assign reasonable names to areas of computer memory; you can use names such as `hoursWorked` or `payRate`, rather than having to remember the memory locations (switch numbers) of those values.

Camel casing, or **lower camel casing**, describes the style of identifiers like `hoursWorked` and `payRate` that appear to have a hump in the middle because they start with a lowercase letter but contain uppercase letters to identify new words. By convention, C# programmers use camel casing when creating variable names. Also, the C# programming language is case sensitive. Therefore, if you create an identifier named `payRate`, you cannot refer to it later using identifiers such as `PayRate` or `payrate`.

Each high-level language has its own **syntax**, or rules of the language. For example, to produce output, you might use the verb *print* in one language and *write* in another. All languages have a specific, limited vocabulary, along with a set of rules for using that vocabulary. Programmers use a computer program called a **compiler** to translate their high-level language statements into machine code. The compiler issues an error message each time a programmer commits a **syntax error**—that is, each time the programmer uses the language incorrectly. Subsequently, the programmer can correct the error and attempt another translation by compiling the program again. The program can be completely translated to machine language only when all syntax errors have been corrected. When you learn a computer programming language such as C#, C++, Visual Basic, or Java, you must learn the vocabulary and syntax rules for that language.

In some languages, such as BASIC, the language translator is called an interpreter. In others, such as assembly language, it is called an assembler. The various language translators operate differently, but the ultimate goal of each is to translate the higher-level language into machine language.

In addition to learning the correct syntax for a particular language, a programmer must understand computer programming logic.

The **logic** behind any program involves executing the various statements and procedures in the correct order to produce the desired results. For example, you might be able to execute perfect individual notes on a musical instrument, but if you do not execute them in the proper order (or execute a B-flat when an F-sharp was expected), no one will enjoy your performance. Similarly, you might be able to use a computer language's syntax correctly but be unable to obtain correct results because the program is not constructed logically. Examples of logical errors include multiplying two values when you should divide them, or attempting to calculate a paycheck before obtaining the appropriate payroll data. The logic used to solve a problem might be identical in two programs, but the programs can be written in different languages, each using different syntax.

Since the early days of computer programming, program errors have been called **bugs**. The term is often said to have originated from an actual moth that was discovered trapped in the circuitry of a computer at Harvard University in 1945. Actually, the term *bug* was in use prior to 1945 to mean trouble with any electrical apparatus; even during Thomas Edison's life, it meant an "industrial defect." In any case, the process of finding and correcting program errors has come to be known as **debugging the program**.

TWO TRUTHS & A LIE

Programming

Two of the following statements are true, and one is false. Identify the false statement, and explain why it is false.

1. A high-level programming language allows you to use a vocabulary of reasonable terms such as *read*, *write*, or *add* instead of the sequence of on/off switches that perform these tasks.
2. Each high-level programming language has its own syntax.
3. Programmers use a computer program called a compiler to translate machine code into a high-level language they can understand.

The false statement is #3. Programmers use a compiler to translate their high-level language statements into machine code.

Procedural and Object-Oriented Programming

Two popular approaches to writing computer programs are procedural programming and object-oriented programming.

When you write a **procedural program,** you use your knowledge of a programming language to create and name computer memory locations that can hold values, and you write a series of steps or operations to manipulate those values. For example, a simple payroll program might contain instructions similar to the following:

```
get hoursWorked
pay = hoursWorked * 10.00
output pay
```

Named computer memory locations such as `hoursWorked` and `pay` are called **variables** because they hold values that might vary. In programming languages, a variable is referenced by using a one-word name (an **identifier)** with no embedded spaces. For example, the memory location referenced by the name `hoursWorked` might contain different values at different times for different employees. During the execution of the payroll program, each value stored under the name `hoursWorked` might have many operations performed on it—for example, reading it from an input device, multiplying it by a pay rate, and printing it on paper.

Examples of procedural programming languages include C and Logo.

For convenience, the individual operations used in a procedural program often are grouped into logical units called **methods**. For example, a series of four or five comparisons and calculations that together determine an employee's withholding tax value might be grouped as a method named `CalculateWithholdingTax()`.

Capitalizing the first letter of all new words in an identifier, even the first one, as in `CalculateWithholdingTax()`, is a style called **Pascal casing** or **upper camel casing**. Although it is legal to start a method name with a lowercase letter, the convention used in C# is for methods to be named using Pascal casing. This helps distinguish them from variables which conventionally use lower camel casing. Additionally, in C# all method names are followed by a set of parentheses. When this book refers to a method, the name will be followed with parentheses.

A procedural program divides a problem solution into multiple methods, each with a unique name. The program then **calls** or **invokes** the methods to input, manipulate, and output the values stored in those locations. A single procedural program often contains hundreds of variables and thousands of method calls.

Depending on the programming language, methods are sometimes called *procedures*, *subroutines*, *or functions*. In C#, the preferred term is *methods*.

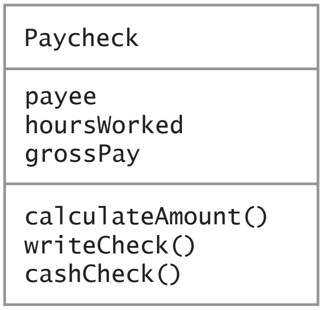


Figure 1-1 A diagram for a `Paycheck`

Object-oriented programming (OOP) is an extension of procedural programming. OOP uses variables and methods like procedural programs do, but it focuses on objects. An **object** is a concrete entity that has attributes and behaviors. The **attributes of an object** are the features it "has"; the values of an object's attributes constitute the **state of the object**. For example, attributes of a paycheck include its payee and monetary value, and the state of those attributes might be *Alice Nelson* and *$400*. The **behaviors of an object** are the things it "does," or its methods; for example, a paycheck object can be written and cashed, and contains a method to calculate the check amount. Figure 1-1 shows how a programmer might start to visualize a paycheck—thinking about its name, attributes, and behaviors. Beyond a paycheck, object-oriented programmers might design a payroll system by thinking about all the additional objects needed to produce a paycheck, such as employees, time cards, and bank accounts.

Programmers use the term *OO*, pronounced "oh oh," as an abbreviation for *object oriented*. When discussing object-oriented programming, they use *OOP*, which rhymes with *soup*.

Examples of OO languages include C#, Java, Visual Basic, and C++. You can write procedural programs in OO languages, but you cannot write OO programs in procedural languages.

With either approach, procedural or object-oriented, you can produce a correct paycheck, and both techniques employ reusable program modules. The major difference lies in the focus the programmer takes during the earliest planning stages of a project. Taking an **object-oriented approach** to a problem means defining the objects needed to accomplish a task and developing classes that describe the objects so that each object maintains its own data and carries out tasks when another object requests them. The object-oriented approach is said to be "natural"—it is more common to think of a world of objects and the ways they interact than to consider a world of systems, data items, and the logic required to manipulate them.

Object-oriented programming employs a large vocabulary; you can learn much of this terminology in the chapter called "Using Classes and Objects."

Originally, object-oriented programming was used most frequently for two major types of applications:

- **Computer simulations** are programs that attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate.
- **Graphical user interfaces, or GUIs** (pronounced "gooeys") are programs that allow users to interact with a program in a graphical environment, such as by clicking with a mouse or using a touch screen.

Thinking about objects in these two types of applications makes sense. For example, a city might want to develop a program that simulates traffic patterns to better prevent congestion. By creating a model with objects such as cars and pedestrians that contain their own data and rules for behavior, the simulation can be set in motion. For example, each car object has a specific current speed and a procedure for changing that speed. By creating a model of city traffic using objects, a computer can create a simulation of a real city at rush hour.

Creating a GUI environment for users also is a natural use for object orientation. It is easy to think of the components a user manipulates on a computer screen, such as buttons and scroll bars, as similar to real-world objects. Each GUI object contains data— for example, a button on a screen has a specific size and color. Each object also contains behaviors—for example, each button can be clicked and reacts in a specific way when clicked. Some people consider the term *object-oriented programming* to be synonymous with GUI programming, but object-oriented programming means more. Although many GUI programs are object oriented, one does not imply the other. Modern businesses use object-oriented design techniques when developing all sorts of business applications, whether they are GUI applications or not.

TWO TRUTHS & A LIE

Procedural and Object-Oriented Programming

1. Procedural programs use variables and tasks that are grouped into methods or procedures.
2. Object-oriented programming languages do not support variables or methods; instead they focus on objects.
3. Object-oriented programs were first used for simulations and GUI programs.

The false statement is #2. Object-oriented programs contain variables and methods just as procedural programs do.

Features of Object-Oriented Programming Languages

For a language to be considered object-oriented, it must support the following features:

- Classes
- Objects
- Encapsulation and interfaces
- Inheritance
- Polymorphism

A **class** describes potential objects, including their attributes and behaviors. A class is similar to a recipe or a blueprint in that it describes what features objects will have and what they will be able to do after they are created. An object is an **instance of a class;** it is one tangible example of a class.

For example, you might create a class named `Automobile`. Some of an `Automobile`'s attributes are its make, model, year, and purchase price. All `Automobile`s possess the same attributes, but not the same values, or states, for those attributes. (Programmers also call the values of an object's attributes the **properties** of the object.) When you create specific `Automobile` objects, each object can hold unique values for the attributes, such as *Ford*, *Taurus*, *2016*, and *$27,000.* Similarly, a `Dog` has attributes that include its breed, name, age, and vaccination status; the attributes for a particular dog might be *Labrador retriever*, *Murphy*, *7*, and *current.*

When you understand that an object belongs to a specific class, you know a lot about the object. If your friend purchases an `Automobile`, you know it has *some* model name; if your friend gets a `Dog`, you know it has *some* breed. You probably don't know the current state of the `Automobile`'s speed or of the `Dog`'s shots, but you do know that those attributes exist for the `Automobile` and

Dog classes. Similarly, in a GUI operating environment, you expect each window you open to have specific, consistent attributes, such as a menu bar and a title bar, because each window includes these attributes as a member of the general class of GUI windows.

By convention, programmers using C# begin their class names with an uppercase letter and use a singular noun. Thus, the class that defines the attributes and methods of an automobile would probably be named `Automobile`, and the class that contains dogs would probably be named `Dog`. If the class requires two words, programmers conventionally use upper camel casing, as in `BankAccount`.

Besides attributes, objects possess methods that they use to accomplish tasks, including changing attributes and discovering the values of attributes. `Automobile`s, for example, have methods for moving forward and backward. They also can be filled with gasoline or be washed; both are methods that change some of an `Automobile`'s attributes. Methods also exist for determining the status of certain attributes, such as the current speed of an `Automobile` and the number of gallons of gas in its tank. Similarly, a `Dog` can walk or run, eat, and get a bath, and there are methods for determining whether it needs a walk, food, or a bath. GUI operating system components, such as windows, can be maximized, minimized, and dragged; depending on the component, they can also have their color or font style altered.

Like procedural programs, object-oriented programs have variables (attributes) and procedures (methods), but the attributes and methods are encapsulated into objects that are then used much like real-world objects. **Encapsulation** is the technique of packaging an object's attributes and methods into a cohesive unit that can be used as an undivided entity. Programmers sometimes refer to encapsulation as using a "**black box**," a device you use without regard for the internal mechanisms. If an object's methods are well written, the user is unaware of the low-level details of how the methods are executed; in such a case, the user must understand only the **interface** or interaction between the method and object. For example, if you can fill your `Automobile` with gasoline, it is because you understand the interface between the gas pump nozzle and the vehicle's gas tank opening. You don't need to understand how the pump works or where the gas tank is located inside your vehicle. If you can read your speedometer, it does not matter how the display value is calculated. In fact, if someone produces a new, more accurate speedometer and inserts it into your `Automobile`, you don't have to know or care how it operates, as long as the interface remains the same as the previous one. The same principles apply to well-constructed objects used in object-oriented programs.

Object-oriented programming languages support two other distinguishing features in addition to organizing objects as members of classes. One feature, **inheritance**, provides the ability to extend a class so as to create a more specific class. The more specific class contains all the attributes and methods of the more general class and usually contains new attributes or methods as well. For example, if you have created a `Dog` class, you might then create a more specific class named `ShowDog`. Each instance of the `ShowDog` class would contain, or inherit, all the attributes and methods of a `Dog`, along with additional methods or attributes. For example, a `ShowDog` might require an attribute to hold the number of ribbons won and a method for entering a dog show. The advantage of inheritance is that when you need a class such as `ShowDog`, you often can extend an existing class, thereby saving a lot of time and work.

Object-oriented languages also support **polymorphism**, which is the ability to create methods that act appropriately depending on the context. That is, programs written in object-oriented languages can distinguish between methods with the same name based on the type of object that uses them. For example, you are able to "fill" both a `Dog` and an `Automobile`, but you do so by very different means. Similarly, the procedure to "fill" a `ShowDog` might require different food than that for a "plain" `Dog`. Older, non-object-oriented languages could not make such distinctions, but object-oriented languages can.

The chapters "Using Classes and Objects" and "Introduction to Inheritance" contain much more information on the features of object-oriented programs.

Watch the video *Object-Oriented Programming*.

TWO TRUTHS & A LIE

Features of Object-Oriented Programming Languages

1. Object-oriented programs contain classes that describe the attributes and methods of objects.
2. Object-oriented programming languages support inheritance, which refers to the packaging of attributes and methods into logical units.
3. Object-oriented programming languages support polymorphism, which is the ability of a method to act appropriately based on the context.

The false statement is #2. Inheritance is the ability to extend classes to make more specific ones. Encapsulation refers to the packaging of attributes and methods.

The C# Programming Language

The **C# programming language** was developed as an object-oriented and component-oriented language. It is part of Microsoft Visual Studio, a package designed for developing applications that run on Windows computers. Unlike other programming languages, C# allows every piece of data to be treated as an object and to consistently employ the principles of object-oriented programming. C# provides constructs for creating components with properties, methods, and events, making it an ideal language for modern programming, where building small, reusable components is more important than building huge, stand-alone applications. You can find Microsoft's C# specifications at *msdn.microsoft.com*. Search for *C# specifications.*

If you have not programmed before, the differences between C# and other languages mean little to you. However, experienced programmers will appreciate the thought that was put into C# features. For example:

- C# contains a GUI interface that makes it similar to Visual Basic, but C# is considered more concise than Visual Basic.
- C# is modeled after the C++ programming language, but is considered easier to learn. Some of the most difficult features to understand in C++ have been eliminated in C#.

Some differences between C# and C++ are that pointers (variables that hold memory addresses) are not used in C# (except in a mode called unsafe, which is rarely used), object destructors and forward declarations are not needed, and using `#include` files is not necessary. Multiple inheritance, which causes many C++ programming errors, is not allowed in C#.

- C# is very similar to Java, because Java was also based on C++. However, C# is more truly object oriented. Unlike in Java, every piece of data in C# is an object, providing all data with increased functionality.

In Java, simple data types are not objects; therefore, they do not work with built-in methods. Additionally, in Java, data can only be passed to and from methods using a copy; C# omits this limitation. You will learn more in two later chapters: "Introduction to Methods" and "Advanced Method Concepts."

The C# programming language was standardized in 2002 by Ecma International. You can read or download this set of standards at *www.ecma-international.org/publications/standards/Ecma-334.htm.*

TWO TRUTHS & A LIE

The C# Programming Language

1. The C# programming language was developed as an object-oriented and component-oriented language.
2. C# contains several features that make it similar to other languages such as Java and Visual Basic.
3. C# contains many advanced features, so the C++ programming language was created as a simpler version of the language.

The false statement is 3. C# is modeled after the C++ programming language, but some of the most difficult features to understand in C++ have been eliminated in C#.

Writing a C# Program that Produces Output

At first glance, even the simplest C# program involves a fair amount of confusing syntax. Consider the simple program in Figure 1-2. This program is written on seven lines, and its only task is to display *This is my first C# program* on the screen.

```
class FirstClass
{
   static void Main()
   {
      System.Console.WriteLine("This is my first C# program");
   }
}
```

Figure 1-2 `FirstClass` console application

The statement that does the actual work in this program is in the middle of the figure:

```
System.Console.WriteLine("This is my first C# program");
```

The statement ends with a semicolon because all C# statements do.

The text `"This is my first C# program"` is a **literal string** of characters—that is, a series of characters that will be used exactly as entered. Any literal string in C# appears between double quotation marks.

The string `"This is my first C# program"` appears within parentheses because the string is an argument to a method, and arguments to methods always appear within parentheses. **Arguments** represent information that a method needs to perform its task. For example, if making an appointment with a dentist's office was a C# method, you would write the following:

```
MakeAppointment("September 10", "2 p.m.");
```

Accepting and processing a dental appointment is a method that consists of a set of standard procedures. However, each appointment requires different information—the date and time—and this information can be considered the arguments of the `MakeAppointment()` method. If you make an appointment for September 10 at 2 p.m., you expect different results than if you make one for September 9 at 8 a.m. or December 25 at midnight. Likewise, if you pass the argument `"Happy Holidays"` to a method, you will expect different results than if you pass the argument `"This is my first C# program"`.

Although an argument to a method might be a string, not all arguments are strings. In this book, you will see and write methods that accept many other types of data.

Within the statement `System.Console.WriteLine("This is my first C# program");`, the method to which you are passing the argument string is named `WriteLine()`. The

`WriteLine()`method displays output on the screen and positions the cursor on the next line, where additional output might be displayed subsequently. The `Write()` method is very similar to the `WriteLine()` method. With `WriteLine()`, the cursor is moved to the following line after the message is displayed. With `Write()`, the cursor does not advance to a new line; it remains on the same line as the output.

Within the statement `System.Console.WriteLine("This is my first C# program");`, `Console` is a class that contains the `WriteLine()` method. Of course, not all classes have a `WriteLine()` method (for instance, you can't write a line to a computer's mouse, an `Automobile`, or a `Dog`), but the creators of C# assumed that you frequently would want to display output on the screen at your terminal. For this reason, the `Console` class was created and endowed with the method named `WriteLine()`. When you use the `WriteLine()` method, programmers say that you *call* it or *invoke* it. Soon, you will create your own C# classes and endow them with your own callable methods.

Within the statement `System.Console.WriteLine("This is my first C# program");`, `System` is a namespace. A **namespace** is a construct that acts like a container to provide a way to group similar classes. To organize your classes, you can (and will) create your own namespaces. The `System` namespace, which is built into your C# compiler, holds commonly used classes.

An advantage to using Visual Studio is that all of its languages use the same namespaces. In other words, everything you learn about any namespace in C# is knowledge you can transfer to Visual C++ and Visual Basic.

The dots (periods) in the phrase `System.Console.WriteLine` are used to separate the names of the namespace, class, and method. You will use this same namespace-dot-class-dot-method format repeatedly in your C# programs.

In the `FirstClass` class in Figure 1-2, the `WriteLine()` statement appears within a method named `Main()`. Every executable C# application must contain a `Main()` method because that is the starting point for every program. As you continue to learn C# from this book, you will write applications that contain additional methods. You will also create classes that are not programs, and so do not need a `Main()` method.

Every method in C# contains a header and a body. A **method header** includes the method name and information about what will pass into and be returned from a method. A **method body** is contained within a pair of curly braces (`{ }`) and includes all the instructions executed by the method. The program in Figure 1-2 includes only one statement between the curly braces of the `Main()` method. Soon, you will write methods with many more statements. In Figure 1-2, the `WriteLine()`statement within the `Main()` method is indented within the curly braces. Although the C# compiler does not require such indentation, it is conventional and clearly shows that the `WriteLine()` statement lies within the `Main()` method.

For every opening curly brace (`{`) in a C# program, there must be a corresponding closing curly brace (`}`). The precise position of the opening and closing curly braces is not important to the compiler. In general, whitespace is optional in C#. **Whitespace** is any combination of spaces, tabs, and carriage returns (blank lines). You use whitespace to organize your program code and make it easier to read; it does not affect your program's execution. Usually, you vertically align each pair of opening and closing curly braces and indent the contents between them, as shown in Figure 1-2.

The method header for the `Main()` method in Figure 1-2 contains three words. Two of these words are keywords. In this book, C# keywords appear in bold. A complete list of keywords appears in Table 1-1 later in this chapter. In the method header `static void Main()`, the keyword **`static`** indicates that the `Main()` method will be executed through a class—not by a variety of objects. It means that you do not need to create an object of type `FirstClass` to use the `Main()` method defined within `FirstClass`. Later in this book, you will create other methods that are nonstatic methods and that are executed by objects.

The second word in the method header in Figure 1-2 is `void`. In English, the word *void* means empty or having no effect. When the keyword **`void`** is used in the `Main()` method header, it does not indicate that the `Main()` method is empty, or that it has no effect, but rather that the method does not return any value when called. You will learn more about methods that return values (and do affect other methods) when you study methods in greater detail.

In the method header, the name of the method is `Main()`. `Main()` is not a C# keyword, but all C# applications must include a method named `Main()`, and most C# applications will have additional methods with other names. Recall that when you execute a C# application, the `Main()` method always executes first. Classes that contain a `Main()` method are **application classes**. Applications are executable or **runnable**. Classes that do not contain a `Main()` method are **non-application classes**, and are not runnable. Non-application classes provide support for other classes.

Watch the video *The Parts of a C# Program.*

TWO TRUTHS & A LIE

Writing a C# Program that Produces Output

1. Strings are information that methods need to perform their tasks.
2. The `WriteLine()` method displays output on the screen and positions the cursor on the next line, where additional output might be displayed.
3. Many methods such as `WriteLine()` have been created for you because the creators of C# assumed you would need them frequently.

The false statement is #1. Strings are literal values represented between quotation marks. Arguments represent information that a method needs to perform its task. Although an argument might be a string, not all arguments are strings.

Selecting Identifiers

Every method that you use within a C# program must be part of a class. To create a class, you use a class header and curly braces in much the same way you use a header and braces for a method within a class. When you write `class FirstClass`, you are defining a class named `FirstClass`. A class name does not have to contain the word *Class* as `FirstClass` does; as a matter of fact, most class names you create will not contain *Class.* You can define a C# class using any identifier you need, as long as it meets the following requirements:

- An identifier must begin with an underscore, the "at" sign (@), or a letter. Letters include foreign-alphabet letters such as Π and Ω, which are contained in the set of characters known as Unicode. You will learn more about Unicode in the next chapter.
- An identifier can contain only letters, digits, underscores, and the @ sign. An identifier cannot contain spaces or any other punctuation or special characters such as #, $, or &.
- An identifier cannot be a C# reserved keyword, such as `class` or `void`. Table 1-1 provides a complete list of reserved keywords. Actually, you can use a keyword as an identifier if you precede it with an @ sign, as in `@class`. An identifier with an @ prefix is a **verbatim identifier**. This feature allows you to use code written in other languages that do not have the same set of reserved keywords. However, when you write original C# programs, you should not use the keywords as identifiers.

`abstract`	`float`	`return`
`as`	`for`	`sbyte`
`base`	`foreach`	`sealed`
`bool`	`goto`	`short`
`break`	`if`	`sizeof`
`byte`	`implicit`	`stackalloc`
`case`	`in`	`static`
`catch`	`int`	`string`
`char`	`interface`	`struct`
`checked`	`internal`	`switch`
`class`	`is`	`this`
`const`	`lock`	`throw`
`continue`	`long`	`true`
`decimal`	`namespace`	`try`
`default`	`new`	`typeof`

Table 1-1 C# reserved keywords *(continues)*

(continued)

`delegate`	`null`	`uint`
`do`	`object`	`ulong`
`double`	`operator`	`unchecked`
`else`	`out`	`unsafe`
`enum`	`override`	`ushort`
`event`	`params`	`using`
`explicit`	`private`	`virtual`
`extern`	`protected`	`void`
`false`	`public`	`volatile`
`finally`	`readonly`	`while`
`fixed`	`ref`	

Table 1-1 C# reserved keywords

The following identifiers have special meaning in C# but are not keywords: `add`, `alias`, `get`, `global`, `partial`, `remove`, `set`, `value`, `where`, and `yield`. For clarity, you should avoid using these words as your own identifiers.

Table 1-2 lists some valid and conventional class names you might use when creating classes in C#. You should follow established conventions for C# so that other programmers can interpret and follow your programs. Table 1-3 lists some class names that are valid, but unconventional; Table 1-4 lists some illegal class names.

Class Name	Description
`Employee`	Begins with an uppercase letter
`FirstClass`	Begins with an uppercase letter, contains no spaces, and has an initial uppercase letter that indicates the start of the second word
`PushButtonControl`	Begins with an uppercase letter, contains no spaces, and has an initial uppercase letter that indicates the start of all subsequent words
`Budget2016`	Begins with an uppercase letter and contains no spaces

Table 1-2 Some valid and conventional class names in C#

Class Name	Description
`employee`	Unconventional as a class name because it begins with a lowercase letter
`First_Class`	Although legal, the underscore is not commonly used to indicate new words in class names
`Pushbuttoncontrol`	No uppercase characters are used to indicate the start of a new word, making the name difficult to read
`BUDGET2016`	Unconventional as a class name because it contains all uppercase letters
`Void`	Although this identifier is legal because it is different from the keyword `void`, which begins with a lowercase *v*, the similarity could cause confusion

Table 1-3 Some unconventional (though legal) class names in C#

Class Name	Description
`an employee`	Space character is illegal
`Push Button Control`	Space characters are illegal
`class`	`class` is a reserved word
`2016Budget`	Class names cannot begin with a digit
`phone#`	The # symbol is not allowed; identifiers consist of letters, digits, underscores, or @

Table 1-4 Some illegal class names in C#

In Figure 1-2, the line `class FirstClass` contains the keyword `class`, which identifies `FirstClass` as a class.

The simple program shown in Figure 1-2 has many pieces to remember. For now, you can use the program shown in Figure 1-3 as a shell, where you replace the identifier `AnyLegalClassName` with any legal class name, and the line `/*********/` with any statements that you want to execute.

```
class AnyLegalClassName
{
   static void Main()
   {
      /*********/;
   }
}
```

Figure 1-3 Shell program

Watch the video *C# Identifiers*.

TWO TRUTHS & A LIE

Selecting Identifiers

1. In C#, an identifier must begin with an underscore, the at sign (@), or an uppercase letter.
2. An identifier can contain only letters, digits, underscores, and the @ sign, not special characters such as #, $, or &.
3. An identifier cannot be a C# reserved keyword.

The false statement is #1. In C#, an identifier must begin with an underscore, the @ sign, or a letter. There is no requirement that the initial letter be capitalized, although in C#, it is a convention that the initial letter of a class name is capitalized.

Improving Programs by Adding Comments and Using the `System` Namespace

As you can see, even the simplest C# program takes several lines of code and contains somewhat perplexing syntax. Large programs that perform many tasks include much more code. As you work through this book, you will discover many ways to improve your ability to handle large programs. Two things you can do immediately are to add program comments and use the `System` namespace.

Adding Program Comments

As you write longer programs, it becomes increasingly difficult to remember why you included steps and how you intended to use particular variables. **Program comments** are nonexecuting statements that you add to document a program. Programmers use comments to leave notes for themselves and for others who might read their programs.

As you work through this book, you should add comments as the first few lines of every program file. The comments should contain your name, the date, and the name of the program. Your instructor might want you to include additional comments.

Comments also can be useful when you are developing a program. If a program is not performing as expected, you can **comment out** various statements and subsequently run the program to observe the effect. When you comment out a statement, you turn it into a comment so the compiler will ignore it. This approach helps you pinpoint the location of errant statements in malfunctioning programs.

C# offers three types of comments:

- **Line comments** start with two forward slashes (`//`) and continue to the end of the current line. Line comments can appear on a line by themselves, or they can occupy part of a line following executable code.
- **Block comments** start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). Block comments can appear on a line by themselves, on a line before executable code, or after executable code. When a comment is long, block comments can extend across as many lines as needed.
- C# also supports a special type of comment used to create documentation within a program. These comments, called **XML-documentation format comments**, use a special set of tags within angle brackets (`< >`). (XML stands for Extensible Markup Language.) You will learn more about this type of comment as you continue your study of C#.

The forward slash (`/`) and the backslash (`\`) characters often are confused, but they are distinct characters. You cannot use them interchangeably.

Figure 1-4 shows how comments can be used in code. The program covers 10 lines, yet only seven are part of the executable C# program, including the last two lines, which contain curly braces and are followed by partial-line comments. The only line that actually *does* anything visible when the program runs is the shaded one that displays *Message*.

```
/* This program is written to demonstrate using comments
*/
class ClassWithOneExecutingLine
{
   static void Main()
   {
      // The next line writes the message
      System.Console.WriteLine(“Message");
   } // End of Main
} // End of ClassWithOneExecutingLine
```

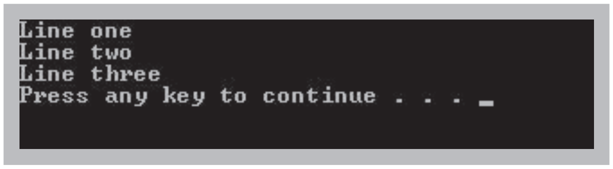
Figure 1-4 Using comments within a program

Using the `System` Namespace

A program can contain as many statements as you need. For example, the program in Figure 1-5 produces the three lines of output shown in Figure 1-6. (To get the output, you have to know how to compile and execute the program, which you will learn in the next part of this chapter.) A semicolon separates each program statement.

```
class ThreeLinesOutput
{
   static void Main()
   {
      System.Console.WriteLine("Line one");
      System.Console.WriteLine("Line two");
      System.Console.WriteLine("Line three");
   }
}
```

Figure 1-5 A program that produces three lines of output



```
Line one
Line two
Line three
Press any key to continue . . . _
```

Figure 1-6 Output of the ThreeLinesOutput program

Figure 1-6 shows the output of the ThreeLinesOutput program when it is run in Visual Studio. The prompt to press any key to continue is not part of the program; it is added by Visual Studio, but it does not appear if you run the program from the command prompt.

The program in Figure 1-5 shows a lot of repeated code—the phrase `System.Console.WriteLine` appears three times. When you need to repeatedly use a class from the same namespace, you can shorten the statements you type by adding a clause that indicates a namespace containing the class. You indicate a namespace with a **using clause**, or **using directive**, as shown in the shaded statement in the program in Figure 1-7. If you type `using System;` prior to the class definition, the compiler knows to use the `System` namespace when it encounters the `Console` class. The output of the program in Figure 1-7 is identical to that in Figure 1-5, in which `System` was repeated with each `WriteLine()` statement.

```
using System;
class ThreeLinesOutput
{
   static void Main()
   {
      Console.WriteLine("Line one");
      Console.WriteLine("Line two");
      Console.WriteLine("Line three");
   }
}
```

Figure 1-7 A program that produces three lines of output with a `using System;` clause

Starting with C# 6.0, you can reduce typing in programs even further by inserting `using static System.Console;` at the top of a program. So, In Visual Studio 2015, the program in Figure 1-8 uses only the method name `WriteLine()` in its output statements, and the program works correctly, (You have already seen the word `static` in the `Main()` method header. Recall that `static` means a method is used without creating an object. In Chapter 7, you will learn more about the use of the keyword `static`.) The simpler style shown in Figure 1-8 is the style used for programs in the rest of this book.

```
using static System.Console;
class ThreeLinesOutput
{
   static void Main()
   {
      WriteLine("Line one");
      WriteLine("Line two");
      WriteLine("Line three");
   }
}
```

Figure 1-8 A program that produces three lines of output with a `using static System.Console;` clause

TWO TRUTHS & A LIE

Improving Programs by Adding Comments and Using the `System` Namespace

1. Line comments start with two forward slashes (//) and end with two backslashes (\\ ).
2. Block comments can extend across as many lines as needed.
3. You use a namespace with a `using` clause, or `using` directive, to shorten statements when you need to repeatedly use a class from the same namespace.

The false statement is #1. Line comments start with two forward slashes (//) and continue to the end of the current line.

You Do It

Now that you understand the basic framework of a program written in C#, you are ready to enter your first C# program into a text editor so you can compile and execute it. It is a tradition among programmers that the first program you write in

(continues)

(continued)

any language produces *Hello, world!* as its output. To create a C# program, you can use any simple text editor, such as Notepad, or the editor that is included as part of Microsoft Visual Studio. There are advantages to using the C# editor to write your programs, but using a plain text editor is simpler when you are getting started.

Entering a Program into an Editor

1. Start any text editor, such as Notepad, and open a new document, if necessary.
2. Type the `using` statement and the header for the class:

   ```
   using static System.Console;
   class Hello
   ```
3. On the next two lines, type the class-opening and class-closing curly braces: **{}**. Some programmers type a closing brace as soon as they type the opening one to guarantee that they always type complete pairs.
4. Between the class braces, insert a new line, type three spaces to indent, and write the `Main()` method header:

   ```
   static void Main()
   ```
5. On the next two lines, type the opening and closing braces for the `Main()` method, indenting them about three spaces.
6. Between the `Main()` method's braces, insert a new line and type six spaces so the next statement will be indented within the braces. Type the one executing statement in this program:

   ```
   WriteLine("Hello, world!");
   ```

   Your code should look like Figure 1-9.

```
using static System.Console;
class Hello
{
   static void Main()
   {
      WriteLine("Hello, world!");
   }
}
```

Figure 1-9 The `Hello` class

(continues)

(continued)

7. Choose a location that is meaningful to you to save your program. For example, you might create a folder named C# on your hard drive. Within that folder, you might create a folder named Chapter.01 in which you will store all the examples and exercises in this chapter. If you are working in a school lab, you might be assigned a storage location on your school's server. Save the program as **Hello.cs**. It is important that the file extension be .cs, which stands for *C Sharp*. If the file has a different extension, the compiler for C# will not recognize the program as a C# program.

Many text editors attach their own filename extension (such as .txt or .doc) to a saved file. Double-check your saved file to ensure that it does not have a double extension (as in Hello.cs.txt). If the file has a double extension, rename it. If you use a word-processing program as your editor, select the option to save the file as a plain text file.

Compiling and Executing a C# Program

After you write and save a program, two more steps must be performed before you can view the program output:

1. You must compile the program you wrote (called the **source code**) into **intermediate language (IL)**.
2. The C# **just in time (JIT)** compiler must translate the intermediate code into executable code.

When you compile a C# program, your source code is translated into intermediate language. The JIT compiler converts IL instructions into native code at the last moment, and appropriately for each type of operating system on which the code might eventually be executed. In other words, the same set of IL can be JIT compiled and executed on any supported architecture.

Some developers say that languages like C# are "semi-compiled." That is, instead of being translated immediately from source code to their final executable versions, programs are compiled into an intermediate version that is later translated into the correct executable statements for the environment in which the program is running.

You can write a program using a simple editor such as Notepad and then perform these steps from the command prompt in your system. You also can write a program within the Integrated Development Environment that comes with Visual Studio. Both methods can produce the same output; the one you use is a matter of preference.

- The **command line** is the line on which you type a command in a system that uses a text interface. The **command prompt** is a request for input that appears at the beginning of the command line. In DOS, the command prompt indicates the disk drive and optional path, and ends with >. You might prefer the simplicity of the command line because you do not work with multiple menus and views. Additionally, if you want to pass command-line arguments to a program, you must compile from the command line.
- The **Integrated Development Environment (IDE)** is a programming environment that allows you to issue commands by selecting choices from menus and clicking buttons. Many programmers prefer using the IDE because it provides features such as color-coded keywords and automatic statement completion.

Compiling Code from the Command Prompt

If you will be using the IDE to write all your programs, you can read this section quickly, and then concentrate on the section titled "Compiling Code Using the Visual Studio IDE."

To compile your source code from the command line, you first locate the developer command prompt. In a Windows 8.1 system, you can swipe from the right, click **Search**, and enter **Developer Command Prompt**. (Depending on your version of Visual Studio, the command prompt might be followed by a year.) If you don't find the command prompt window, you need to obtain and install a copy of Visual Studio. For more information, visit *msdn2.microsoft.com/en-us/vcsharp.*

In the developer command prompt window, you type **`csc`**, followed by the name of the file that contains the source code. The command `csc` stands for *C Sharp compiler*. For example, to compile a file named ThreeLinesOutput.cs, you would type the following and then press **Enter**:

```
csc ThreeLinesOutput.cs
```

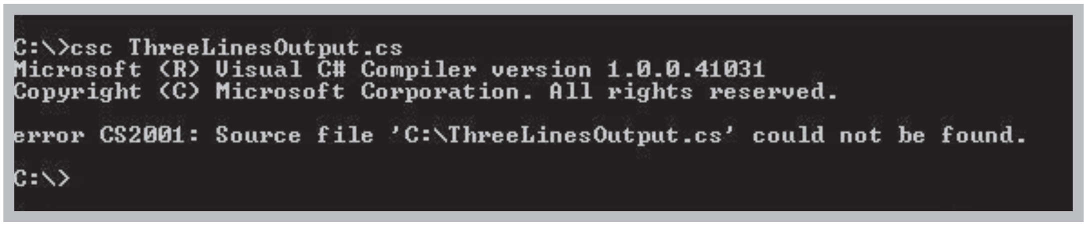
One of three outcomes is possible:

- You receive an operating system error message such as *Bad command or file name* or *csc is not recognized as an internal or external command, operable program or batch file*. You can recognize operating system messages because they do not start with the name of the program you are trying to compile.
- You receive one or more program language error messages. You can recognize program language error messages because they start with the name of the program followed by a line number and the position where the error was first noticed.
- You receive no error messages (only a copyright statement from Microsoft), indicating that the program has compiled successfully.

What to Do If You Receive an Operating System Error Message at the Command Prompt

If you receive an operating system message such as *csc is not recognized...*, or *Source file... could not be found*, it may mean that:

- You misspelled the command `csc`.
- You misspelled the filename.
- You forgot to include the extension .cs with the filename.
- You are not within the correct subdirectory or folder on your command line. For example, Figure 1-10 shows the `csc` command typed in the root directory of the C: drive. If the ThreeLinesOutput.cs file is stored in a folder on the C: drive rather than in the root directory, then the command shown will not work because the C# file cannot be found.
- The C# compiler was not installed properly. If you are working on your own computer, you might need to reinstall C#; if you are working in a school laboratory, you need to notify the system administrator.

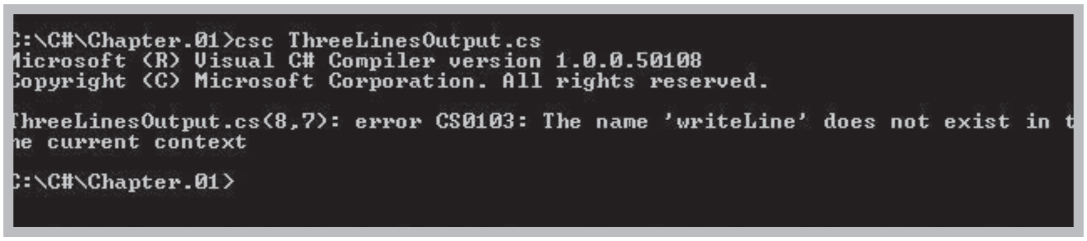


```
C:\>csc ThreeLinesOutput.cs
Microsoft (R) Visual C# Compiler version 1.0.0.41031
Copyright (C) Microsoft Corporation. All rights reserved.

error CS2001: Source file 'C:\ThreeLinesOutput.cs' could not be found.

C:\>
```

Figure 1-10 Attempt to compile a program from the root directory at the command prompt, and error message received

What to Do If You Receive a Programming Language Error Message at the Command Prompt

If you receive a programming language error message, it means that the compiler was installed correctly, but that the source code contains one or more syntax errors. A syntax error occurs when you introduce typing errors into your program. Program error messages start with the program name, followed by parentheses that hold the line number and the position in the line where the compiler noticed the error. For example, in Figure 1-11, an error is found in ThreeLinesOutput.cs in line 8, position 7. In this case, the message is generated because `WriteLine` is typed as *writeLine* (with a lowercase *w*). The error message is *The name 'writeLine" does not exist in the current context.* If a problem like this occurs, you must reopen the text file that contains the source code, make the necessary corrections, save the file, and compile it again.



```
:\C#\Chapter.01>csc ThreeLinesOutput.cs
icrosoft (R) Visual C# Compiler version 1.0.0.50108
opyright (C) Microsoft Corporation. All rights reserved.

hreeLinesOutput.cs(8,7): error CS0103: The name 'writeLine' does not exist in t
e current context

:\C#\Chapter.01>
```

Figure 1-11 Error message generated when `WriteLine` is mistyped in the ThreeLinesOutput program compiled from the command prompt

The C# compiler issues warnings as well as errors. A warning is less serious than an error; it means that the compiler has determined you have done something unusual, but not illegal. If you have purposely introduced a warning situation to test a program, then you can ignore the warning. Usually, however, you should treat a warning message just as you would an error message and attempt to remedy the situation.

What to Do When the Program Compiles Successfully at the Command Prompt

If you receive no error messages after compiling the code, then the program compiled successfully, and a file with the same name as the source code—but with an .exe extension—is created and saved in the same folder as the program text file. For example, if ThreeLinesOutput.cs compiles successfully, then a file named ThreeLinesOutput.exe is created.

To run the program from the command line, you simply type the program name—for example, *ThreeLinesOutput*. You can also type the full filename, *ThreeLinesOutput.exe*, but it is not necessary to include the extension. The three lines of output, *Line one*, *Line two*, and *Line three*, appear in the command prompt window, and the command prompt is displayed again, awaiting a new command.

Compiling Code Using the Visual Studio IDE

As an alternative to using the command line, you can compile and write your program within the Visual Studio IDE. This approach has several advantages:

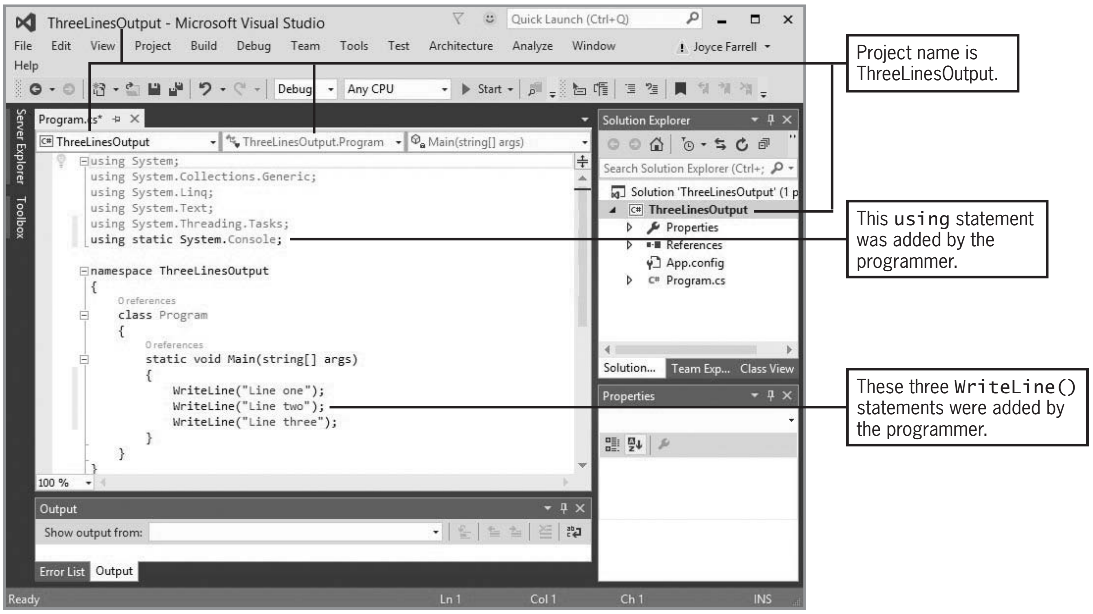
- Some of the code you need is already created for you.
- The code is displayed in color, so you can more easily identify parts of your program. Reserved words appear in blue, comments in green, and identifiers in black.
- If error messages appear when you compile your program, you can double-click an error message and the cursor will move to the line of code that contains the error.
- Other debugging tools are available. You will become more familiar with these tools as you develop more sophisticated programs.

If Visual Studio has been installed on your system, you can open it from the **menu bar** which, in the Visual Studio IDE, is the list of choices that runs horizontally across the top of the screen.

Start a new project, designate it to be a Console Project, and give the project a name. Figure 1-12 shows a program and project named ThreeLinesOutput written in the editor of the Visual Studio IDE. Everything in the figure was generated automatically by Visual Studio except for the following:

- The programmer selected the project name.
- The programmer added a `using static System.Console;` statement so that `WriteLine()` could be used without adding `System` and `Console` each time.
- The programmer added the three lines coded in the `Main()` method to produce output.

You can see that the Visual Studio environment looks like a word processor, containing menu options such as File, Edit, and Help, and buttons with icons representing options such as Save Selected Items and Undo. Selecting many of the menu options displays a drop-down list of additional options. Many of the options have shortcut keys that you can use instead of accessing the menus. For example, if you select File from the menu bar, you can see that a shortcut for Save Program.cs is Ctrl + S. You will learn about some of these options later in this chapter and continue to learn about more of them as you work with C# in the IDE.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;

namespace ThreeLinesOutput
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            WriteLine("Line one");
            WriteLine("Line two");
            WriteLine("Line three");
        }
    }
}
```

ThreeLinesOutput - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Architecture Analyze Window Joyce Farrell
Help
Debug Any CPU Start
Program.cs*
ThreeLinesOutput
ThreeLinesOutput.Program
Main(string[] args)
Server Explorer Toolbox
100 %
Output
Show output from:
Error List Output
Ready Ln 1 Col 1 Ch 1 INS
Solution Explorer
Search Solution Explorer (Ctrl+;
Solution 'ThreeLinesOutput' (1 p
ThreeLinesOutput
Properties
References
App.config
Program.cs
Solution... Team Exp... Class View
Properties
Project name is ThreeLinesOutput.
This using statement was added by the programmer.
These three WriteLine() statements were added by the programmer.

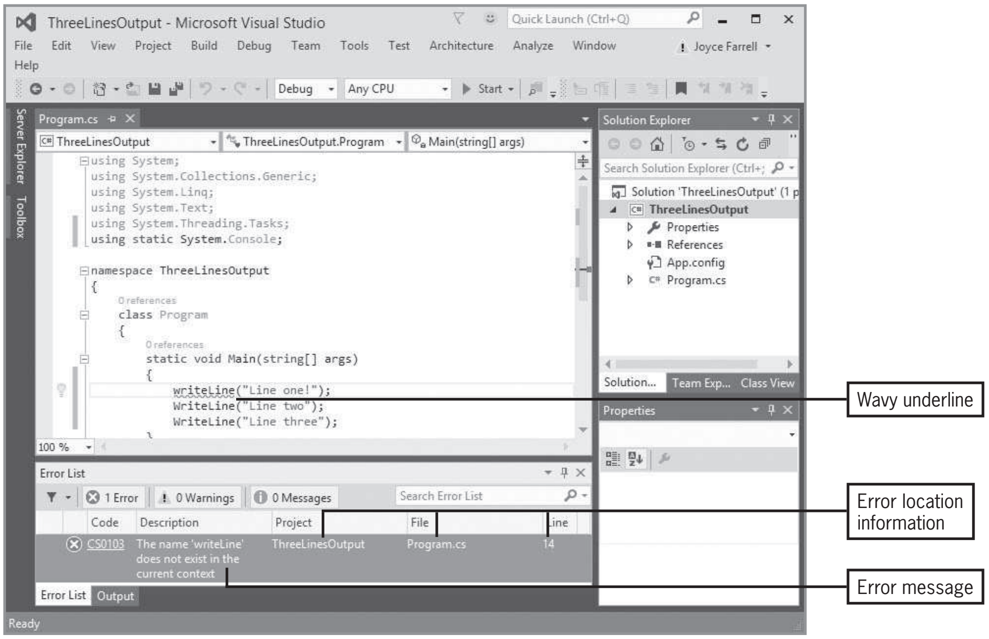
Figure 1-12 The ThreeLinesOutput program as it appears in Visual Studio

In Figure 1-12 you can see two instances of *0 references*, above the class header and the `Main()` method header. This means that neither this class nor method is used by any other method. When you start to write methods in Chapter 7, you will be able to see how many places your methods are referenced.

One way to compile a program from Visual Studio is to select **Build** from the menu bar and then select Build Solution. As an alternative, you can press Ctrl + Shift + B. You can also select **Debug** from the menu bar, and then click **Start Without Debugging**. The advantage of the latter option is that the program will be compiled if there are no syntax errors and then execute so you can see the output.

What to Do If You Receive a Programming Language Error Message in the IDE

If you introduce a syntax error into a program in the IDE, you receive a programming language error message. For example, in Figure 1-13, `WriteLine` is spelled incorrectly because it uses a lowercase *w*. The error message displayed at the bottom is: *'The name 'writeLine" does not exist in the current context'*. (If you cannot read the entire message, you can adjust the size of the window.) The position is given (project, file, and line number), and as an additional visual aid, a wavy underline emphasizes the unrecognized definition. When you fix the problem and compile again, the error message is eliminated.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;

namespace ThreeLinesOutput
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            writeLine("Line one!");
            WriteLine("Line two");
            WriteLine("Line three");
```

Code	Description	Project	File	Line
CS0103	The name 'writeLine' does not exist in the current context	ThreeLinesOutput	Program.cs	14

Figure 1-13 Error message generated when `WriteLine` is mistyped in the ThreeLinesOutput program compiled in the IDE

What to Do When the Program Compiles Successfully in the IDE

In Figure 1-13 you can see a code CS0103 next to the generated error message. You can search the web for this code and discover its general meaning. However, usually sufficient explanation is provided in Visual Studio, and a web search is unnecessary.

If you receive no error messages after compiling the code, then the program compiled successfully, and you can run the program. Select **Debug** from the menu bar, and then select **Start Without Debugging**. The output appears, as you saw it in Figure 1-6 earlier in this chapter.

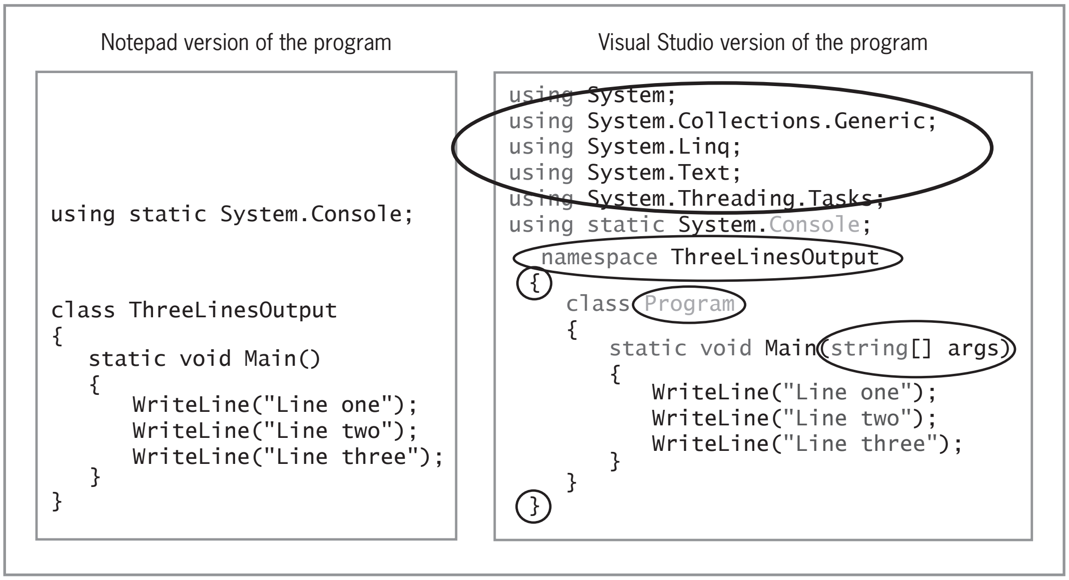
Noticing the Differences Between the Programs in the Text Editor and the IDE

There are a few differences between the ThreeLinesOutput programs in Figures 1-8 and 1-12. Specifically, the Visual Studio version in Figure 1-12 contains the following extra components that are highlighted in Figure 1-14:

- *Five `using` statements at the top of the file*—Visual Studio automatically provides you with several commonly needed `using` statements. In this case, you could delete all five statements from the Visual Studio version of the program, and the program would still run correctly. (The only `using` statement you need to retain is the `using static System.Console;` statement that allows you to use `WriteLine()` without qualification.) Conversely, you could add all five additional `using` statements to the command-line version of the program, and it would also run correctly.
- *A namespace declaration and its opening and closing curly braces*—You already know that C# uses built-in namespaces like `System` and others that are automatically listed at the top of the Visual Studio version of programs. C# also allows you to create your own namespaces; Visual Studio assumes that you will want to create one using the name you have assigned to the project. In this case, you could remove the namespace declaration from the Visual Studio version of the program, or you could add one to the command-line version of the program. In both cases, the programs would still run correctly.
- *The class name Program*—Visual Studio assumes that you want to avoid confusion by having different names for your namespace and the class contained within it, so it assigns the generic name `Program` to the class that contains the `Main()` method. You could change this name to match the namespace identifier, `ThreeLinesOutput`, or use any other legal identifier, and the program would still work correctly. In the Notepad version of the program, you could change the class name to `Program` or any other legal identifier, and the program would also work.
- *The words* `string[] args` *between the parentheses of the* `Main()` *method header.* C# allows a program's `Main()` method header to be written in several ways. For example, you can include the words `string[] args` between the method's parentheses, or you can omit

the words. The square brackets indicate an array, as you will learn when studying arrays later in this book. In this example, you could write the `Main()` method header with or without the extra code in either the Notepad or Visual Studio version of the program, and the `Main()` method would still work correctly.

In summary, the Visual Studio version of the ThreeLinesOutput program provides everything you need to run the program and saves you as much typing as possible in case you want to include additional options. The command-line and Visual Studio versions of the program function identically.


Notepad version of the program
Visual Studio version of the program

```
using static System.Console;


class ThreeLinesOutput
{
   static void Main()
   {
      WriteLine("Line one");
      WriteLine("Line two");
      WriteLine("Line three");
   }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
namespace ThreeLinesOutput
{
   class Program
   {
      static void Main(string[] args)
      {
         WriteLine("Line one");
         WriteLine("Line two");
         WriteLine("Line three");
      }
   }
}
```

Figure 1-14 Comparing the command line and Visual Studio versions of the ThreeLinesOutput program

Deciding Which Environment to Use

When you write, compile, and execute a C# program, you can use either a text editor and the command line or the Visual Studio IDE. You would never need to use both. You might prefer using an editor with which you are already familiar (such as Notepad) and compiling from the command line because only two files are generated, saving disk space.

On the other hand, the IDE provides many useful features, such as automatic statement completion. For example, if you type *u*, then a list of choices including `using` is displayed and you can click using instead of typing it. Similarly, if you type *using s*, then a list of choices including `static` is displayed

Additionally, in the IDE, words are displayed using different colors based on their category; for example, one color is used for C#-reserved words and a different color for literal strings. It is also easier to correct many errors using the IDE. When compiler errors or warnings are issued, you can double-click the message, and the cursor jumps to the location in the code where the error was detected. Another advantage to learning the IDE is that if you use another programming language in Visual Studio (C++ or Visual Basic), the environment will already be familiar to you.

The C# language works the same way no matter which method you use to compile your programs. Everything you learn in the next chapters about input, output, decision making, loops, and arrays is correct in both environments. You can use just one technique or compile some programs in each environment as the mood strikes you. You can also mix and match techniques if you prefer. For example, you can use an editor you like to compose your programs, then paste them into the IDE to execute them. Although any program can be written using either compilation technique, when you write GUI applications that use existing objects such as buttons and labels, you will find that the extensive amount of code automatically generated by the IDE is very helpful.

Watch the video Writing and Compiling a Program.

TWO TRUTHS & A LIE

Compiling and Executing a C# Program

1. After you write and save a program, you must compile it into intermediate language and then the C# just in time (JIT) compiler must translate the intermediate code into executable code.
2. You can compile and execute a C# program from the command line or within the Integrated Development Environment (IDE) that comes with Visual Studio.
3. Many programmers prefer to compile their programs from the command line because it provides features such as color-coded keywords and automatic statement completion.

The false statement is #3. Programmers who prefer the command line prefer its simplicity. Programmers who prefer the Visual Studio IDE prefer the color-coded keywords and automatic statement completion.

You Do It

Compiling and Executing a Program from the Command Line

If you do not plan to use the command line to execute programs, you can skip to the next part of this "You Do It" section, called "Compiling and Executing a Program Using the Visual Studio IDE."

1. Go to the command prompt on your system. For example, in Windows 8.1, swipe from the right, click **Search**, start to type **Developer Command Prompt**, and then click it.
2. Change the current directory to the name of the folder that holds your program. You can type `cd\` and then press Enter to return to the root directory. You can then type `cd` to change the path to the one where your program resides. For example, if you stored your program file in a folder named Chapter.01 within a folder named C#, then you can type the following:

   ```
   cd C#\Chapter.01
   ```

   The command `cd` is short for *change directory.*
3. Type the command that compiles your program:

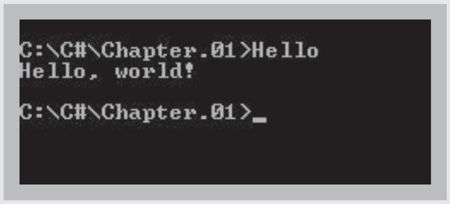
   ```
   csc Hello.cs
   ```

   If you receive no error messages and the prompt returns, it means that the compile operation was successful, that a file named Hello.exe has been created, and that you can execute the program. If you do receive error messages, check every character of the program you typed to make sure it matches Figure 1-9 in the last "You Do It" section. Remember, C# is case sensitive, so all casing must match exactly. When you have corrected the errors, repeat this step to compile the program again.
4. You can verify that a file named Hello.exe was created in these ways:
   - At the command prompt, type **dir** to view a directory of the files stored in the current folder. Both Hello.cs and Hello.exe should appear in the list.
   - Use Windows Explorer to view the contents of the Chapter.01 folder, verifying that two Hello files are listed.

(continues)

(continued)

5. At the command prompt, type **Hello**, which is the name of the program (the name of the executable file), and then press **Enter**. Alternatively, you can type the full filename **Hello.exe**, but typing the .exe extension isn't necessary. The output should look like Figure 1-15.



```
C:\C#\Chapter.01>Hello
Hello, world!

C:\C#\Chapter.01>_
```

Figure 1-15 Output of the Hello application

You can use the `/out` compiler option between the `cs` command and the name of the .cs file to indicate the name of the output file. For example, if you type `csc /out:Hello.exe Hello.cs`, you create an output file named Hello.exe. By default, the name of the output file is the same as the name of the .cs file. Usually, this is your intention, so most often you omit the `/out` option.

Compiling and Executing a Program Using the Visual Studio IDE

Next, you use the C# compiler environment to compile and execute the same Hello program you ran from the command line.

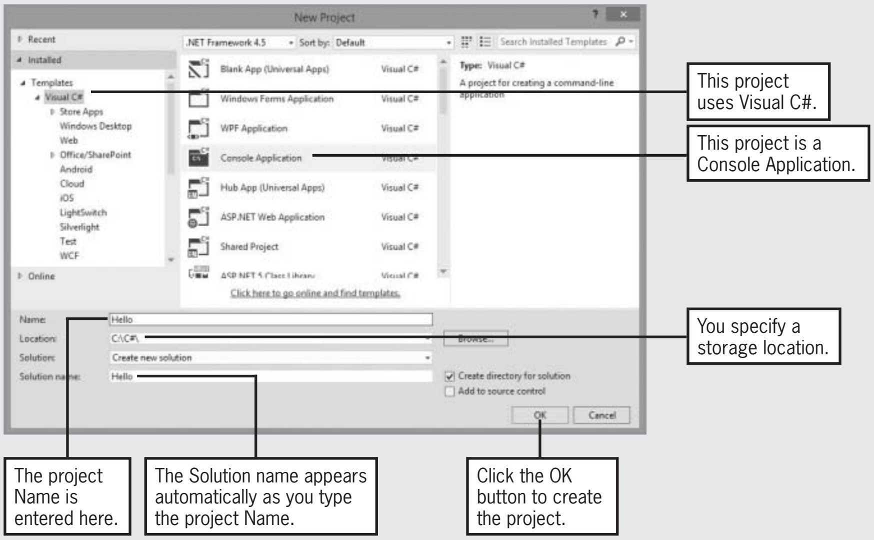
1. Open **Visual Studio**. If there is a shortcut icon on your desktop, you can double-click it. Alternatively, in Windows 8, you can swipe from the right, click **Search**, start to type **Visual Studio 2015**, and select it. When you see the Start Page, click **New Project**.
2. In the New Project window, click **Visual C#**, and then click **Console Application**. By default, the name for the project is set to *ConsoleApplication1*.

   Change it to **Hello**, and select the path where you want to save the project (see Figure 1-16). Click **OK**. Visual C# creates a new folder for your project named after the project title.

Students might like to walk through these "You Do It" exercises multiple times for practice. Note that Visual Studio does not allow you to create a project with the same name as one already stored in a folder. When you perform these steps additional times, either delete the first version of the project before you start the next one, store the new version in a different folder, or give the new project a different name.

(continues)

(continued)


New Project
Recent
Installed
Templates
Visual C#
Store Apps
Windows Desktop
Web
Office/SharePoint
Android
Cloud
iOS
LightSwitch
Silverlight
Test
WCF
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates
Blank App (Universal Apps) Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Hub App (Universal Apps) Visual C#
ASP.NET Web Application Visual C#
Shared Project Visual C#
ASP.NET 5 Class Library Visual C#
Click here to go online and find templates.
Type: Visual C#
A project for creating a command-line application
Browse...
Create directory for solution
Add to source control
OK
Cancel
This project uses Visual C#.
This project is a Console Application.
You specify a storage location.
The project Name is entered here.
The Solution name appears automatically as you type the project Name.
Click the OK button to create the project.

Name: Hello
Location: C:\C#\
Solution: Create new solution
Solution name: Hello

Figure 1-16 Starting a new project

3. The Hello application editing window appears. A lot of code is already written for you in this window, including some `using` statements, a namespace named `Hello`, a class named `Program`, and a `Main()` method. To create the Hello program, you need to add only two statements to the prewritten code. Place your cursor after the last using statement, press **Enter** to start a new line, and add an additional using statement:

```
using static System.Console;
```

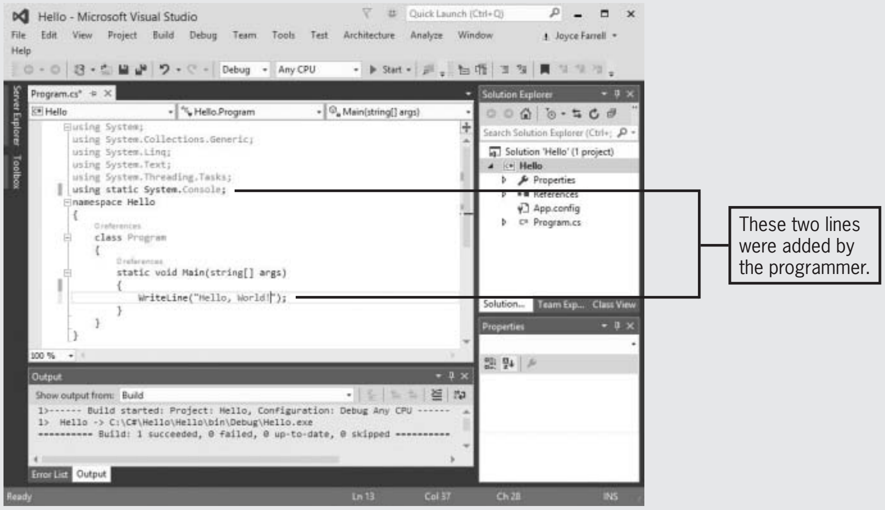
After the opening brace of the `Main()` method, press **Enter** to start a new line, and add the following statement:

```
WriteLine("Hello, world!");
```

Figure 1-17 shows the editing window after the `using` and `WriteLine()` statements have been added.

(continues)

(continued)

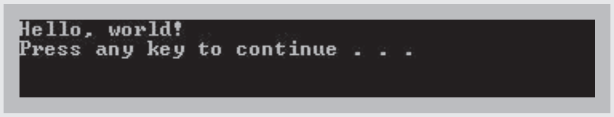

Hello - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Architecture Analyze Window Help
Joyce Farrell
Debug
Any CPU
Start
Program.cs*
Hello
Hello.Program
Main(string[] args)
Solution Explorer
Solution 'Hello' (1 project)
Hello
Properties
References
App.config
Program.cs
Properties
Output
Show output from: Build
Error List
Output
Ready Ln 13 Col 37 Ch 28 INS
These two lines were added by the programmer.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
namespace Hello
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            WriteLine("Hello, World!");
        }
    }
}
```

```
1>------ Build started: Project: Hello, Configuration: Debug Any CPU ------
1>  Hello -> C:\C#\Hello\Hello\bin\Debug\Hello.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 1-17 The Hello application editing window after the code has been added

4. Save the project by clicking **File** on the menu bar and then clicking **Save All**, or by clicking the **Save All** button (which shows two disks) on the toolbar, or by typing **Ctrl + Shift + S**.
5. To compile the program, click **Build** on the menu bar, and then click **Build Solution**. You should receive no error messages, and the words *Build succeeded* should appear near the lower-left edge of the window.
6. Click **Debug** on the menu bar, and then click **Start Without Debugging**. Figure 1-18 shows the output; you see *Hello, world!* followed by the message *Press any key to continue*. Press any key to close the output screen.



```
Hello, world!
Press any key to continue . . .
```

Figure 1-18 Output of the Hello application in Visual Studio

(continues)

(continued)

7. Close Visual Studio by clicking **File** on the menu bar and then clicking **Exit**, or by clicking the **Close** box in the upper-right corner of the Visual Studio window.
8. When you create a C# program using an editor such as Notepad and compile it with the `csc` command, only two files are created—Hello.cs and Hello.exe. When you create a C# program using the Visual Studio editor, many additional files are created. You can view their filenames from the command prompt or visually:
   - At the command prompt, type **dir** to view a directory of the files stored in the folder where you saved the project (for example, your Chapter.01 folder). Within the folder, a new folder named Hello has been created. Type the command **cd Hello** to change the current path to include this new folder, then type **dir** again. You see another folder named Hello. Type **cd Hello** again, and **dir** again. You should see several folders and files. (Depending on the version of Visual Studio you are using, your folder configuration might be slightly different.)
   - Use File Explorer to view the contents of the Hello folders within the Chapter.01 folder.

   Regardless of the technique you use to examine the folders, you will find that the innermost Hello folder contains a bin folder, an obj folder, a Properties folder, and additional files. If you explore further, you will find that the bin folder contains Debug and Release folders, which include additional files. Using the Visual Studio editor to compile your programs creates a significant amount of overhead. These additional files become important as you create more sophisticated C# projects.

If you followed the earlier instructions on compiling a program from the command line, and you used the same folder when using the IDE, you will see the additional Hello.cs and Hello.exe files in your folder. These files will have an earlier time stamp than the files you just created. If you were to execute a new program within Visual Studio without saving and executing it from the command line first, you would not see these two additional files.

(continues)

(continued)

Adding Comments to a Program

Comments are nonexecuting statements that help to document a program. In the following steps, you add some comments to the program you just created.

1. If you prefer compiling programs from the command line, then open the **Hello.cs** file in your text editor. If you prefer compiling programs from Visual Studio, then open Visual Studio, click **File**, point to **Open**, click **Project/ Solution**, browse for the correct folder, double-click the **Hello** folder, and then double-click the **Hello** file.
2. Position your cursor at the top of the file, press **Enter** to insert a new line, press the **up** arrow to go to that line, and then type the following comments at the top of the file. Press **Enter** after typing each line. Insert your name and today's date where indicated.

```
//Filename Hello.cs
//Written by <your name>
//Written on <today's date>
```

3. Scroll to the line that reads `static void Main()`, and press **Enter** to start a new line. Then press the **up** arrow; in the new blank line, aligned with the start of the `Main()` method header, type the following block comment in the program:

```
/*  This program demonstrates the use of the WriteLine()
method to display the message Hello, world!  */
```

4. Save the file, replacing the old Hello.cs file with this new, commented version.
5. If you prefer to compile programs from the command line, type `csc Hello.cs` at the command line. When the program compiles successfully, execute it with the command **Hello**. If you prefer compiling and executing programs from Visual Studio, click **Debug**, and then click **Start Without Debugging**. Adding program comments makes no difference in the execution of the program.
6. If you are working from the command line, you can close the command-line window. If you are working from Visual Studio, you can close the output screen and then close Visual Studio.

Chapter Summary

- A computer program is a set of instructions that tell a computer what to do. Programmers write their programs, then use a compiler to translate their high-level language statements into intermediate language and machine code. A program works correctly when both its syntax and logic are correct.
- Procedural programming involves creating computer memory locations, called variables, and sets of operations, called methods. In object-oriented programming, you envision program components as objects that are similar to concrete objects in the real world; then you manipulate the objects to achieve a desired result. OOP techniques were first used for simulations and GUIs.
- Objects are instances of classes and are made up of attributes and methods. Object-oriented programming languages support encapsulation, inheritance, and polymorphism.
- The C# programming language was developed as an object-oriented and component-oriented language. It contains many features similar to those in Visual Basic, Java, and C++.
- To produce a line of console output in a C# program, you must pass a literal string as an argument to the `System.Console.WriteLine()` method. `System` is a namespace and `Console` is a class. Calls to the `WriteLine()` method can appear within the `Main()` method of a class you create.
- You can define a C# class or variable by using any name or identifier that begins with an underscore, a letter, or an @ sign. These names can contain only letters, digits, underscores, and the @ sign, and cannot be C#-reserved keywords.
- You can improve programs by adding comments, which are nonexecuting statements that you add to document a program or to disable statements when you test a program. The three types of comments in C# are line comments that start with two forward slashes (`//`) and continue to the end of the current line, block comments that start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`), and XML-documentation comments. You can also improve programs and shorten the statements you type by using a clause that indicates a namespace where your classes can be found.
- To create a C# program, you can use the Microsoft Visual Studio IDE or any text editor, such as Notepad. After you write and save a program, you must compile the source code into intermediate and machine language.

Key Terms

A computer **program** is a set of instructions that tell a computer what to do.

Software is computer programs.

System software describes the programs that operate the computer.

Application software is the programs that allow users to complete tasks.

Hardware comprises all the physical devices associated with a computer.

Machine language is the most basic circuitry-level language.

A **high-level programming language** allows you to use a vocabulary of keywords instead of the sequence of on/off switches that perform these tasks.

Keywords are predefined and reserved identifiers that have special meaning to the compiler.

Camel casing, also called **lower camel casing**, is a style of creating identifiers in which the first letter is not capitalized, but each new word is.

A language's **syntax** is its set of rules.

A **compiler** is a computer program that translates high-level language statements into machine code.

A **syntax error** is an error that occurs when a programming language is used incorrectly.

The **logic** behind any program involves executing the various statements and methods in the correct order to produce the desired results.

Semantic errors are the type of logical errors that occur when you use a correct word in the wrong context, generating incorrect results.

A **bug** is an error in a computer program.

Debugging a program is the process of removing all syntax and logical errors from the program.

A **procedural program** is created by writing a series of steps or operations to manipulate values.

Variables are named computer memory locations that hold values that might vary.

An **identifier** is the name of a program component such as a variable, class, or method.

Pascal casing, also called **upper camel casing**, is a style of creating identifiers in which the first letter of all new words in a name, even the first one, is capitalized.

Methods are compartmentalized, named program units containing instructions that accomplish tasks.

A program **calls** or **invokes** methods.

Object-oriented programming (OOP) is a programming technique that features objects, classes, encapsulation, interfaces, polymorphism, and inheritance.

An **object** is a concrete entity that has attributes and behaviors; an object is an instance of a class.

The **attributes of an object** represent its characteristics.

The **state of an object** is the collective value of all its attributes at any point in time.

The **behaviors of an object** are its methods.

Taking an **object-oriented approach** to a problem means defining the objects needed to accomplish a task and developing classes that describe the objects so each maintains its own data and carries out tasks when another object requests them.

Computer simulations are programs that attempt to mimic real-world activities to foster a better understanding of them.

Graphical User Interfaces, or **GUIs** (pronounced "gooeys"), are program elements that allow users to interact with a program in a graphical environment.

A **class** is a category of objects or a type of object.

An **instance of a class** is an object.

The **properties** of an object are its values.

Encapsulation is the technique of packaging an object's attributes and methods into a cohesive unit that can be used as an undivided entity.

A **black box** is a device you use without regard for the internal mechanisms.

An **interface** is the interaction between a method and an object.

Inheritance is the ability to extend a class so as to create a more specific class that contains all the attributes and methods of a more general class; the extended class usually contains new attributes or methods as well.

Polymorphism is the ability to create methods that act appropriately depending on the context.

The **C# programming language** was developed as an object-oriented and component-oriented language. It exists as part of Visual Studio, a package used for developing applications for the Windows family of operating systems.

A **literal string** of characters is a series of characters enclosed in double quotes that is used exactly as entered.

An **argument** to a method represents information that a method needs to perform its task. An argument is the expression used between parentheses when you call a method.

A **namespace** is a construct that acts like a container to provide a way to group similar classes.

A **method header** includes the method name and information about what will pass into and be returned from a method.

The **method body** of every method is contained within a pair of curly braces (`{}`) and includes all the instructions executed by the method.

Whitespace is any combination of spaces, tabs, and carriage returns (blank lines). You use whitespace to organize your program code and make it easier to read.

The keyword `static`, when used in a method header, indicates that a method will be executed through a class and not by an object.

The keyword `void`, when used in a method header, indicates that the method does not return any value.

Application classes contain a `Main()` method and are executable programs.

Runnable describes files that are executable.

Non-application classes do not contain a `Main()` method; they provide support for other classes.

A **verbatim identifier** is an identifier with an @ prefix.

Program comments are nonexecuting statements that you add to document a program.

To **comment out** a statement is to make a statement nonexecuting.

Line comments start with two forward slashes (`//`) and continue to the end of the current line. Line comments can appear on a line by themselves or at the end of a line following executable code.

Block comments start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). Block comments can appear on a line by themselves, on a line before executable code, or after executable code. They can also extend across as many lines as needed.

XML-documentation format comments use a special set of tags within angle brackets to create documentation within a program.

A **using clause** or **using directive** declares a namespace.

Source code is the statements you write when you create a program.

Intermediate language (IL) is the language into which source code statements are compiled.

The C# **just in time (JIT)** compiler translates intermediate code into executable code.

The **command line** is the line on which you type a command in a system that uses a text interface.

The **command prompt** is a request for input that appears at the beginning of the command line.

An **Integrated Development Environment (IDE)** is a program development environment that allows you to select options from menus or by clicking buttons. An IDE provides such helpful features as color coding and automatic statement completion.

The **menu bar** in the IDE lies horizontally across the top of the window, and includes submenus that list additional options.

Review Questions

1. Programming languages such as Java and Visual Basic are ______________________ .
   - a. machine languages
   - b. low-level languages
   - c. high-level languages
   - d. uninterpreted languages

2. A program that translates high-level programs into intermediate or machine code is a(n) ______________________ .
   - a. mangler
   - b. compiler
   - c. analyst
   - d. logician

3. The grammar and spelling rules of a programming language constitute its ______________________ .
   - a. logic
   - b. variables
   - c. syntax
   - d. class

4. Variables are ______________________ .
   - a. methods
   - b. named memory locations
   - c. grammar rules
   - d. operations

5. Programs in which you create and use objects that have attributes similar to their real-world counterparts are known as ______________________ programs.
   - a. procedural
   - b. logical
   - c. authentic
   - d. object-oriented

6. Which of the following pairs is an example of a class and an object, in that order?
   - a. Robin and bird
   - b. Chair and desk
   - c. University and Harvard
   - d. Oak and tree

7. The technique of packaging an object's attributes into a cohesive unit that can be used as an undivided entity is ______________________ .
   - a. inheritance
   - b. encapsulation
   - c. polymorphism
   - d. interfacing

8. Of the following languages, which is least similar to C#?
   - a. Java
   - b. Visual Basic
   - c. C++
   - d. COBOL

9. A series of characters that appears within double quotation marks is a(n)___.
   - a. method
   - b. interface
   - c. argument
   - d. literal string

10. The C# method that produces a line of output on the screen and then positions the cursor on the next line is ______________________ .
    - a. `WriteLine()`
    - b. `PrintLine()`
    - c. `DisplayLine()`
    - d. `OutLine()`

11. Which of the following is a class?
    - a. `System`
    - b. `Console`
    - c. `void`
    - d. `WriteLine()`

12. In C#, a container that groups similar classes is a(n) ______________________ .
    - a. superclass
    - b. method
    - c. namespace
    - d. identifier

13. Every method in C# contains a ______________________ .
    - a. header and a body
    - b. header and a footer
    - c. variable and a class
    - d. class and an object

14. Which of the following is a method?
    - a. `namespace`
    - b. `class`
    - c. `Main()`
    - d. `static`

15. Which of the following statements is true?
    - a. An identifier must begin with an underscore.
    - b. An identifier can contain digits.
    - c. An identifier must be no more than 16 characters long.
    - d. An identifier can contain only lowercase letters.

16. Which of the following identifiers is not legal in C#?
    - a. `per cent increase`
    - b. `annualReview`
    - c. `HTML`
    - d. `alternativetaxcredit`

17. The text of a program you write is called ______________________.
    - a. object code
    - b. source code
    - c. machine language
    - d. executable documentation

18. Programming errors such as using incorrect punctuation or misspelling words are collectively known as ____________________ errors.
    a. syntax
    b. logical
    c. executable
    d. fatal

19. A comment in the form `/* this is a comment */` is a(n)___.
    a. XML comment
    b. block comment
    c. executable comment
    d. line comment

20. If a programmer inserts `using static System.Console;` at the top of a C# program, which of the following can the programmer use as an alternative to `System.Console.WriteLine("Hello");`?
    a. `System("Hello");`
    b. `WriteLine("Hello");`
    c. `Console.WriteLine ("Hello");`
    d. `Console("Hello");`

Exercises

Programming Exercises

1. Indicate whether each of the following C# programming language identifiers is legal or illegal. If it is legal, indicate whether it is a conventional identifier for a class.
    a. `paycheck`
    b. `Paycheck`
    c. `Pay check`
    d. `static`
    e. `Void`
    f. `#phone`
    g. `Ay56we`
    h. `Theater_Tickets`
    i. `102Item`
    j. `heightInInches`
    k. `Zip23891`
    l. `void`

2. Name at least three attributes that might be appropriate for each of the following classes:
    a. `TVShow`
    b. `Party`
    c. `JobApplication`
    d. `CheckingAccount`

3. Name at least two classes to which each of these objects might belong:
    a. your sister
    b. Franklin D. Roosevelt
    c. Starbucks
    d. Paris

4. Write, compile, and test a program named **PersonalInfo** that displays a person's name, e-mail address, and phone number.
5. Write, compile, and test a program named **Lyrics** that displays at least four lines of your favorite song.
6. Write, compile, and test a program named **Comments** that displays a statement that defines program comments. Include at least one block comment and one line comment in the program.
7. Write, compile, and test a program named **WineGlass** that displays a pattern similar to the following on the screen:

```
XXXXXXXXXXXXX
 X         X
  X       X
   XXXXXXX
      X
      X
    XXXXX
```

8. Write a program named **BigLetter** that displays a large letter composed of smaller letters as in the following example:

```
H      H
H      H
H      H
HHHHHHHH
H      H
H      H
H      H
```

9. From 1925 through 1963, Burma Shave advertising signs appeared next to highways all across the United States. There were always four or five signs in a row containing pieces of a rhyme, followed by a final sign that read "Burma Shave." For example, one set of signs that has been preserved by the Smithsonian Institution reads as follows:

   Shaving brushes
   You'll soon see 'em
   On a shelf
   In some museum
   Burma Shave

   Find a classic Burma Shave rhyme on the Web and write a program named **BurmaShave** that displays the rhyme.

Debugging Exercises

1. Each of the following files in the Chapter.01 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, DebugOne1.cs will become FixedDebugOne1.cs.
   a. DebugOne1.cs
   b. DebugOne2.cs
   c. DebugOne3.cs
   d. DebugOne4.cs

Case Problems

The case problems in this section introduce two fictional businesses. Throughout this book, you will create increasingly complex classes for these businesses that use the newest concepts you have mastered in each chapter.

1. Greenville County hosts the Greenville Idol competition each summer during the county fair. The talent competition takes place over a 3-day period during which contestants are eliminated following rounds of performances until the year's ultimate winner is chosen. Write a program named **GreenvilleMotto** that displays the competition's motto, which is "The stars shine in Greenville." Create a second program named **GreenvilleMotto2** that displays the motto surrounded by a border composed of asterisks.

2. Marshall's Murals is a company that paints interior and exterior murals for both business and residential customers. Write a program named **MarshallsMotto** that displays the company motto, which is "Make your vision your view." Create a second program named **MarshallsMotto2** that displays the motto surrounded by a border composed of repeated *M*s.