CHAPTER 2

Using Data

In this chapter you will:

- Learn about declaring variables
- Display variable values
- Learn about the integral data types
- Learn about floating-point data types
- Use arithmetic operators
- Learn about the `bool` data type
- Learn about numeric type conversion
- Learn about the `char` data type
- Learn about the `string` data type
- Define named constants and enumerations
- Accept console input

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

In Chapter 1, you learned about programming in general and the C# programming language in particular. You wrote, compiled, and ran a C# program that produced output. In this chapter, you build on your basic C# programming skills by learning how to manipulate data and use data types, variables, and constants. As you will see, using variables to hold data makes computer programs flexible.

Declaring Variables

A data item is **constant** when it cannot be changed after a program is compiled—in other words, when it cannot vary. For example, if you use the number 347 within a C# program, then 347 is a constant, and every time you execute the program, the value 347 will be used. You can refer to the number 347 as a **literal constant**, because its value is taken literally at each use. (A number like 347 is also called an *unnamed constant* because no name is associated with it. Later in this chapter you will learn about named constants.)

On the other hand, when you want a data item's value to be able to change, you can create a variable. A **variable** is a named location in computer memory that can hold different values at different points in time. For example, if you create a variable named `heatingBill` and include it in a C# program, `heatingBill` might contain the value 347, or it might contain 200. Different values might be used when the program is executed multiple times, or different values might be used at different times during a single program execution. Because you can use a variable to hold `heatingBill` within a program used by a utility company's automated billing system, you can write one set of instructions to compute `heatingBill`, yet use different `heatingBill` values for thousands of utility customers.

Whether it is a constant or a variable, each data item you use in a C# program has a data type. A **data type** describes the format and size of (amount of memory occupied by) a data item and defines what types of operations can be performed with the item. C# provides for 15 basic, or **intrinsic types**, of data, as shown in Table 2-1. Of these built-in data types, the ones most commonly used are `int`, `double`, `decimal`, `char`, `string`, and `bool`. Each C# intrinsic type is an **alias**, or other name, for a class in the `System` namespace. (You learned about the `System` namespace in Chapter 1.) All of the built-in types except `object` and `string` are called **simple types**.

The meaning of the largest and smallest values for the numeric types in Table 2-1 is probably clear to you, but some of the other values require explanation.

- The highest `char` value, 0xFFFF, represents the character in which every bit is turned on. The lowest value, 0x0000, represents the character in which every bit is turned off. Any value that begins with *0x* represents a hexadecimal, or base 16, value.
- For any two `string`s, the one with the higher Unicode character value in an earlier position is considered higher. (**Unicode** is the system that provides a numeric value for every character. You will learn more about Unicode later in this chapter, and Appendix B contains further information.) For example, "AAB" is higher than "AAA". The `string` type has no true minimum; however, you can think of the empty string `""` as being the lowest.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Although the `bool` type has no true maximum or minimum, you can think of `true` as the highest and `false` as the lowest.

Type	System Type	Bytes	Description	Largest Value	Smallest Value
Types that hold numbers with no decimal places					
`byte`	`Byte`	1	Unsigned byte	255	0
`sbyte`	`Sbyte`	1	Signed byte	127	–128
`short`	`Int16`	2	Signed short	32,767	–32,768
`ushort`	`UInt16`	2	Unsigned short	65,535	0
`int`	`Int32`	4	Signed integer	2,147,483,647	–2,147,483,648
`uint`	`UInt32`	4	Unsigned integer	4,294,967,295	0
`long`	`Int64`	8	Signed long integer	Approximately 9×10^{18}	Approximately -9×10^{18}
`ulong`	`UInt64`	8	Unsigned long integer	Approximately 18×10^{18}	0
Types that hold numbers with decimal places					
`float`	`Single`	4	Floating-point	Approximately 3.4×10^{38}	Approximately 3.4×10^{38}
`double`	`Double`	8	Double-precision floating-point	Approximately 1.8×10^{308}	Approximately 1.8×10^{308}
`decimal`	`Decimal`	16	Fixed-precision number	Approximately 7.9×10^{28}	Approximately 7.9×10^{28}
Types that hold other values					
`char`	`Char`	2	Unicode character	0xFFFF	0x0000
`bool`	`Boolean`	1	Boolean value (true or false)	NA	NA
`string`	`String`	NA	Unicode string	NA	NA
`object`	`Object`	NA	Any object	NA	NA

Table 2-1 C# data types

You name variables using the same rules for identifiers as you use for class names. A variable name must start with a letter, cannot include embedded spaces, and cannot be a reserved keyword. (Table 1-1 in Chapter 1 lists all the C# keywords.) By convention, each C# variable name starts with a lowercase letter and uses camel casing if it contains multiple words.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You must declare all variables you want to use in a program. A **variable declaration** is the statement that names a variable and reserves storage for it. A declaration includes

- The data type that the variable will store
- The variable's name (its identifier)
- An optional assignment operator and assigned value when you want a variable to contain an initial value
- An ending semicolon

For example, the following statement declares a variable of type `int` named `myAge` and assigns it an initial value of 25:

```
int myAge = 25;
```

In other words, the statement reserves four bytes of memory with the name `myAge`, and the value 25 is stored there. The declaration is a complete statement that ends in a semicolon.

The equal sign (=) is the **assignment operator**; any value to the right of the assignment operator is assigned to the identifier to the left. An assignment made when a variable is declared is an **initialization**; an assignment made later is simply an **assignment**. Thus, the following statement initializes `myAge` to 25:

```
int myAge = 25;
```

A statement such as the following assigns a new value to the variable:

```
myAge = 42;
```

Notice that the data type is not used again when an assignment is made; it is used only in a declaration.

Also note that the expression `25 = myAge;` is illegal because assignment always takes place from right to left. By definition, a literal constant cannot be altered, so it is illegal to place one (such as 25) on the left side of an assignment operator. The assignment operator means "is assigned the value of the following expression." In other words, the statement `myAge = 25;` can be read as "`myAge` is assigned the value of the following expression: 25."

Instead of using a type name from the Type column of Table 2-1, you can use the fully qualified type name from the `System` namespace that is listed in the System Type column. For example, instead of using the type name `int`, you can use the full name `System.Int32`. (The number 32 in the name `System.Int32` represents the number of bits of storage allowed for the data type. There are eight bits in a byte, and an `int` occupies four bytes.) It's better to use the shorter alias `int`, however, for the following reasons:

- The shorter alias is easier to type and read.
- The shorter alias resembles type names used in other languages such as Java and C++.
- Other C# programmers expect the shorter type names.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You can declare a variable without an initialization value, as in the following statement:

```
int myAge;
```

You can make an assignment to an uninitialized variable later in the program, but you cannot use the variable in a calculation or display it until a value has been assigned to it.

You can declare multiple variables of the same type in separate statements. For example, the following statements declare two integer variables:

```
int myAge = 25;
int yourAge = 19;
```

You also can declare multiple variables of the same type in a single statement by using the type once and separating the variable declarations with a comma, as shown in the following statement:

```
int myAge = 25, yourAge = 19;
```

Some programmers prefer to break declarations across multiple lines, as in the following example:

```
int myAge = 25,
    yourAge = 19;
```

When a statement occupies more than one line, it is easier to read if lines after the first one are indented a few spaces. This book follows that convention.

When declaring variables of different types, you must use a separate statement for each type.

Watch the video *Declaring Variables*.

TWO TRUTHS & A LIE

Declaring Variables

1. A constant value cannot be changed after a program is compiled, but a variable can be changed.
2. A data type describes the format and size of a data item and the types of operations that can be performed with it.
3. A variable declaration requires a data type, name, and assigned value.

The false statement is #3. A variable declaration names a variable and reserves storage for it; it includes the data type that the variable will store and an identifier. An assignment operator and assigned value can be included, but they are not required.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Displaying Variable Values

You can display variable values by using the variable name within a `Write()` or `WriteLine()` method call. For example, Figure 2-1 shows a C# program that displays the value of the variable `someMoney`. Figure 2-2 shows the output of the program when executed at the command prompt.


```
using static System.Console;
class DisplaySomeMoney
{
   static void Main()
   {
      double someMoney = 39.45;
      WriteLine(someMoney);
   }
}
```

Figure 2-1 Program that displays a variable value

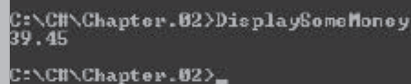

Figure 2-2 Output of the `DisplaySomeMoney` program

The output shown in Figure 2-2 is rather stark—just a number with no explanation. The program in Figure 2-3 adds some explanation to the output; the result is shown in Figure 2-4. This program uses the `Write()` method to display the string "The money is $" before displaying the value of `someMoney`. Because the program uses `Write()` instead of `WriteLine()` for the first output, the second output appears on the same line as the first.

```
using static System.Console;
class DisplaySomeMoney2
{
   static void Main()
   {
      double someMoney = 39.45;
      Write("The money is $");
      WriteLine(someMoney);
   }
}
```

Figure 2-3 Program that displays a string and a variable value

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

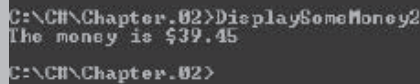

Figure 2-4 Output of the `DisplaySomeMoney2` program

If you want to display several strings and several variables, you can end up with quite a few `Write()` and `WriteLine()` statements. To make producing output easier, you can combine strings and variable values into a single `Write()` or `WriteLine()` statement by concatenating them or by using a format string.

When you **concatenate** a string with another value, you join the values with a plus sign. For example, the following statement produces the same output as shown in Figure 2-4 but uses only one `WriteLine()` statement:

```
WriteLine("The money is $" + someMoney);
```

A **format string** is a string of characters that controls the appearance of output. It optionally contains fixed text and contains one or more format items or *placeholders* for variable values. A **place holder** consists of a pair of curly braces containing a number that indicates the desired variable's position in a list that follows the string. The first position is always position 0.

You can create a formatted string using the `String.Format()` method that is built into C#. For example, the following statements create a formatted string and then display it:

```
double someMoney = 39.45;
string mySentence =
   String.Format("The money is ${0} exactly", someMoney);
WriteLine(mySentence);
```

The curly braces that contain a digit create a placeholder `{0}` into which the value of `someMoney` is inserted. Because `someMoney` is the first variable after the format string (as well as the only variable in this example), its position is 0. The process of replacing string place holders with values is often called *string interpolation* or *variable substitution*.

C# also allows you to use a shortcut with the `Write()` and `WriteLine()` methods. You can format a string and output it in one step without declaring a temporary string variable and without explicitly naming the `String.Format()` method, as in the following:

```
double someMoney = 39.45;
WriteLine("The money is ${0} exactly", someMoney);
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

For example, the program in Figure 2-5 produces the output shown in Figure 2-6.

```
using static System.Console;
class DisplaySomeMoney3
{
   static void Main()
   {
      double someMoney = 39.45;
      WriteLine("The money is ${0} exactly", someMoney);
   }
}
```

Figure 2-5 Using a format string

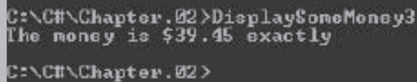

Figure 2-6 Output produced using format string

To display two variables within a single call to `Write()` or `WriteLine()`, you can use a statement like the following:

```
WriteLine("The money is {0} and my age is {1}", someMoney, myAge);
```

In this example, the value of `someMoney` will appear at position 0, and `myAge` will appear at position 1. The number within any pair of curly braces in a format string must be less than the number of values you list after the format string. In other words, if you list six values to be displayed, valid format position numbers are 0 through 5. You do not have to use the positions in order. You also can display a specific value multiple times. For example, if `someMoney` has been assigned the value 439.75, the following code produces the output shown in Figure 2-7:

```
WriteLine("I have ${0}. ${0}!! ${0}!!", someMoney);
```

Figure 2-7 Displaying the same value multiple times

The output in Figure 2-7 could be displayed without using a format string, but you would have to name the `someMoney` variable three separate times as follows:

```
WriteLine("I have $" + someMoney + ". $" + someMoney +
   "!! $" + someMoney + "!!");
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

A new way to display concatenated strings is introduced in C# 6.0. With this method you can use variable names within curly braces that are preceded with a slash. For example, the following code produces the output *Money is 42.56 and age is 29*.

```
double someMoney = 42.56;
int myAge = 29;
WriteLine("Money is \{someMoney} and age is \{myAge}");
```

When you use a series of `WriteLine()` statements to display a list of variable values, the values are not automatically right-aligned as you normally expect numbers to be. For example, the following code produces the unaligned output shown in Figure 2-8:

```
WriteLine("{0}", 4);
WriteLine("{0}", 56);
WriteLine("{0}", 789);
```

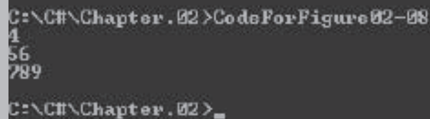

Figure 2-8 Displaying values with different numbers of digits without using field sizes

If you use a second number within the curly braces in a format string, you can specify alignment and field size. By default, numbers are right-aligned in their fields. If you use a negative value for the field size in a `Write()` or `WriteLine()` statement, the displayed value will be left-aligned in the field. Most often, you want numbers to be right-aligned and string values to be left-aligned. For example, the following code produces the output shown in Figure 2-9. The output created by each of the first three `WriteLine()` statements is a right-aligned number in a field that is five characters wide, and the output created by each of the last three statements is two left-aligned strings in a field that is eight characters wide.

```
WriteLine("{0, 5}", 4);
WriteLine("{0, 5}", 56);
WriteLine("{0, 5}", 789);
WriteLine("{0, -8}{1, -8}", "Richard", "Lee");
WriteLine("{0, -8}{1, -8}", "Marcia", "Parker");
WriteLine("{0, -8}{1, -8}", "Ed", "Tompkins");
```

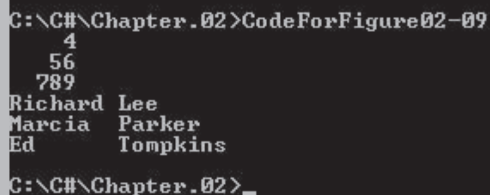

Figure 2-9 Displaying values with different numbers of digits using field sizes

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Format strings can become very long. However, when you include any string value in a program, you cannot extend it across multiple lines by pressing Enter. If you do, you will receive an error message: "Newline in constant". Instead, if you want to break a long string into multiple lines of code, you can concatenate multiple strings into a single entity, as in the following:


```
WriteLine("I have ${0}. " +
   "${0} is a lot. ", someMoney);
```

In the concatenated example, the + could go at the end of the first code line or the beginning of the second one. If the + is placed at the end of the first line, someone reading your code is more likely to notice that the statement is not yet complete. Because of the limitations of this book's page width, you will see examples of concatenation frequently in program code.

TWO TRUTHS & A LIE

Displaying Variable Values

1. Assuming `number` and `answer` are legally declared variables with assigned values, then the following statement is valid:

```
WriteLine("{1}{2}", number, answer);
```

2. Assuming `number` and `answer` are legally declared variables with assigned values, then the following statement is valid:

```
WriteLine("{1}{1}{0}", number, answer);
```

3. Assuming `number` and `answer` are legally declared variables with assigned values, then the following statement is valid:

```
WriteLine("{0}{1}{1}{0}", number, answer);
```

The false statement is #1. When two values are available for display, the position number between any pair of curly braces in the format string must be 0 or 1. If three values were listed after the format string, the position numbers could be any combination of 0, 1, and 2.

Using the Integral Data Types

In C#, nine data types are considered **integral data types**—that is, types that store whole numbers. The nine types are **`byte`**, **`sbyte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, and **`char`**. The first eight always represent whole numbers, and the ninth type, `char`, is used for characters like *A* or *a*. Actually, you can think of all nine types as numbers because every Unicode character, including the letters of the alphabet and punctuation marks, can be represented as a number. For example, the character *A* is stored within your computer as a 65. Because you more commonly think of the `char` type as holding alphabetic characters instead of their numeric equivalents, the `char` type will be discussed in its own section later in this chapter.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The most basic of the other eight integral types is `int`. You use variables of type `int` to store (or hold) **integers**, or whole numbers. An `int` uses four bytes of memory and can hold any whole number value ranging from 2,147,483,647 down to –2,147,483,648. When you type a whole-number constant such as 3 or 589 into a C# program, by default it is an `int`.

If you want to save memory and know you need only a small value, you can use one of the shorter integer types—`byte`, `sbyte` (which stands for signed byte), `short` (short `int`), or `ushort` (unsigned short `int`). For example, a payroll program might contain a variable named `numberOfDependents` that is declared as type `byte`, because `numberOfDependents` will never need to hold a value exceeding 255; for that reason, you can allocate just one byte of storage to hold the value. When you declare variables, you always make a judgment about which type to use. If you use a type that is too large, you waste storage. If you use a type that is too small, your program won't compile. Many programmers simply use `int` for most whole numbers.

Saving memory is seldom an issue for an application that runs on a PC. However, when you write applications for small devices with limited memory, such as smartphones, conserving memory becomes more important.

The word *integer* has only one *r*—at the end. Pronouncing it as "in-ter-ger," with an r in the middle, would mark you as unprofessional.

When you assign a value to any numeric variable, you do not use any commas or other nonnumeric characters such as dollar or percent signs; you type only digits. You can also type a plus or minus sign to indicate a positive or negative integer; a value without a sign is assumed to be positive. For example, to initialize a variable named `annualSalary`, you might write the following without a dollar sign or comma:

```
int annualSalary = 20000;
```

TWO TRUTHS & A LIE

Using the Integral Data Types

1. C# supports nine integral data types, including the most basic one, `int`.
2. Every Unicode character, including the letters of the alphabet and punctuation marks, can be represented as a number.
3. When you assign a value to any numeric variable, it is optional to use commas for values in the thousands.

The false statement is #3. When you assign a value to a numeric variable, you do not use commas, but you can type a plus or minus sign to indicate a positive or negative integer.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You Do It

Declaring and Using Variables

In the following steps, you write a program that declares several integral variables, assigns values to them, and displays the results.

1. Open a new file. If you are using a simple text editor to write your C# programs, you will save the file as **DemoVariables.cs**. If you are using Visual Studio, select **Console Application**, name the project **DemoVariables**, and delete all the code in the program-editing window before starting.
2. Create the beginning of a program that will demonstrate variable use. Name the class `DemoVariables`, and type the class-opening curly brace.

```
using static System.Console;
class DemoVariables
{
```

3. In the `Main()` method, declare two variables (an integer and an unsigned integer), and assign values to them.

```
static void Main()
{
   int anInt = -123;
   uint anUnsignedInt = 567;
```

4. Add a statement to display the two values.

```
WriteLine("The int is {0} and the unsigned int is {1}.",
   anInt, anUnsignedInt);
```

5. Add two closing curly braces—one that closes the `Main()` method and one that closes the `DemoVariables` class. Align each closing curly brace vertically with the opening brace that is its partner. In other words, the first closing brace aligns with the brace that opens `Main()`, and the second aligns with the brace that opens `DemoVariables`.
6. Save the program, and compile it. If you receive any error messages, correct the errors and compile the program again. When the file is error-free, execute the program. The output should look like Figure 2-10.

```
C:\C#\Chapter.02>DemoVariables
The int is -123 and the unsigned int is 567.

C:\C#\Chapter.02>_
```

Figure 2-10 Output of the `DemoVariables` program

(continues)

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

7. Experiment with the program by introducing invalid values for the named variables. For example, change the value of `anUnsignedInt` to **–567** by typing a minus sign in front of the constant value. Compile the program. You receive the following error message: *Constant value '–567' cannot be converted to a 'uint'.*
8. Correct the error either by removing the minus sign or by changing the data type of the variable to **`int`**, and compile the program again. You should not receive any error messages. Remember to save your program file after you make each change and before you compile.
9. Change the value of `anInt` from **–123** to **–123456789000**. When you compile the program, the following error message appears: *Cannot implicitly convert type 'long' to 'int'.*

   The value is a `long` because it is greater than the highest allowed `int` value. Correct the error either by using a lower value or by changing the variable type to `long`, and compile the program again. You should not receive any error messages.
10. Experiment with other changes to the variables. Include some variables of type `short`, `ushort`, `byte`, and `sbyte`, and experiment with their values.

Using Floating-Point Data Types

A **floating-point** number is one that contains decimal positions. Floating-point numbers are described as having varying numbers of significant digits. A value's number of **significant digits** specifies the mathematical accuracy of the value. The concept is often used in rounding. For example, in the statement "The population of New York City is 8,000,000," there is just one significant digit—the 8. The zeros serve as placeholders and are meant to approximate the actual value.

When numbers are stored in computer memory, using a decimal point that can "float" provides a wider possible range of values. As a very simple example, suppose that the largest number you could store had only two digits with a fixed decimal point between them. Then the positive numbers that could be stored would range from 0.0 to 9.9. However, if the decimal point can float to be first, middle, or last, then the range becomes .00 through 99.

C# supports three floating-point data types: `float`, `double`, and `decimal`.

- A **`float`** data type can hold as many as seven significant digits of accuracy. For example, a `float` assigned the value 1234.56789 will appear as 1234.568 because it is accurate only to the seventh digit (the third digit to the right of the decimal point).
- A **`double`** data type can hold 15 or 16 significant digits of accuracy. For example, a `double` given the value 123456789.987654321 will appear as 123456789.987654 because it is accurate to only the fifteenth digit (the sixth digit to the right of the decimal point).

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Compared to `float`s and `double`s, the **`decimal`** type has a greater precision and a smaller range, which makes it suitable for financial and monetary calculations. A `decimal` is significant to 28 or 29 digits. As Table 2-1 shows, a `decimal` cannot hold as large a value as a `double` can (the highest value is about 10^{28}; the highest value of a `double` is 10^{308}), but the `decimal` will be more accurate to more decimal places.

Just as an integer constant is an `int` by default, a floating-point number constant is a `double` by default. To explicitly store a constant as a `float`, you place an *F* after the number, as in the following:

```
float pocketChange = 4.87F;
```

You can use either a lowercase or uppercase *F*. You can also place a *D* (or *d*) after a floating-point value to indicate that it is a `double`; even without the *D*, however, it will be stored as a `double` by default. To explicitly store a value as a `decimal`, use an *M* (or *m*) after the number. (*M* stands for monetary; *D* can't be used for a `decimal` because it indicates `double`.)

Because monetary values are extremely important in business applications, many C# developers frequently use the `decimal` data type. This book most often uses the `double` type for floating-point values because floating-point numbers are `double` by default and do not require a modifying letter after the value. However, you should use the data type preferred by your instructor or manager or the type that is most appropriate to your application. Some programmers prefer to use the `decimal` type for any measurement created by humans (for example, money or test scores) and `double` for measurements of naturally occurring phenomena (for example, half-lives of elements).

If you store a value that is too large (or too small) in a floating-point variable, you will see output expressed in **scientific notation**. Values expressed in scientific notation include an *E* (for exponent). For example, if you declare `float f = 1234567890f;`, the value is output as 1.234568E9, meaning that the numeric constant you used is approximately 1.234568 times 10 to the ninth power, or 1.234568 with the decimal point moved nine positions to the right. When the value that follows the *E* is negative, it means the value is very small and the decimal point should move to the left. For example, the value 1.234E–07 represents 0.0000001234.

Formatting Floating-Point Values

By default, C# always displays floating-point numbers in the most concise way possible that maintains the correct value. For example, if you declare a variable and display it as in the following statements, the output will appear as *The amount is 14.*

```
double myMoney = 14.00;
WriteLine("The amount is {0}", myMoney);
```

The two zeros to the right of the decimal point in the assigned value will not appear in the output because they add no mathematical information. To see the decimal places, you can convert the floating-point value to a string using a standard numeric format string.

Standard numeric format strings are strings of characters expressed within double quotation marks that indicate a format for output. They take the form *X0*, where *X* is the format specifier

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

and *0* is the precision specifier. The **format specifier** can be one of nine built-in format characters that define the most commonly used numeric format types. The **precision specifier** controls the number of significant digits or zeros to the right of the decimal point. Table 2-2 lists the nine format specifiers.

Format Character	Description	Default Format (if no precision is given)
C or c	Currency	$XX,XXX.XX ($XX,XXX.XX)
D or d	Decimal	[–]XXXXXXX
E or e	Scientific (exponential)	[–]X.XXXXXXE+xxx [–]X.XXXXXXe+xxx [–]X.XXXXXXE-xxx [–]X.XXXXXXe-xxx
F or f	Fixed-point	[–]XXXXXXX.XX
G or g	General	Variable; either with decimal places or scientific
N or n	Number	[–]XX,XXX.XX
P or p	Percent	Represents a numeric value as a percentage
R or r	Round trip	Ensures that numbers converted to strings will have the same values when they are converted back into numbers
X or x	Hexadecimal	Minimum hexadecimal (base 16) representation

Table 2-2 Format specifiers

You can use a format specifier with the `ToString()` method to convert a number into a string that has the desired format. For example, you can use the *F* format specifier to insert a decimal point to the right of a number that does not contain digits to the right of the decimal point, followed by the number of zeros indicated by the precision specifier. (If no precision specifier is supplied, two zeros are inserted.) For example, the first `WriteLine()` statement in the following code produces 123.00, and the second produces 123.000:

```
double someMoney = 123;
string moneyString;
moneyString = someMoney.ToString("F");
WriteLine(moneyString);
moneyString = someMoney.ToString("F3");
WriteLine(moneyString);
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You will learn more about creating and using methods in the chapters "Introduction to Methods" and "Advanced Method Concepts."

You use *C* as the format specifier when you want to represent a number as a currency value. Currency values appear with a dollar sign, appropriate commas, and the desired number of decimal places; negative values appear within parentheses. The integer you use following the *C* indicates the number of decimal places. If you do not provide a value for the number of decimal places, then two digits are shown after the decimal separator by default. For example, the following `WriteLine()` statement produces $456,789.00:

```
double moneyValue = 456789;
string conversion;
conversion = moneyValue.ToString("C");
WriteLine(conversion);
```

Currency appears with a dollar sign and commas in the English culture. A **culture** is a set of rules that determines how culturally dependent values such as money and dates are formatted. You can change a program's culture by using the `CultureInfoClass`. The .NET framework supports more than 200 culture settings, such as Japanese, French, Urdu, and Sanskrit.

To display a numeric value as a formatted string, you do not have to create a separate string object. You also can make the conversion in a single statement; for example, the following code displays $12,345.00:

```
double payAmount = 12345;
WriteLine(payAmount.ToString("C"));
```

TWO TRUTHS & A LIE

Using Floating-Point Data Types

1. A floating-point number is one in which the decimal point varies each time you reference it.
2. C# supports three floating-point data types: `float`, `double`, and `decimal`.
3. To explicitly store a constant as a `float`, you may place an *F* after the number, but to store a constant as a `double` you need no special designation.

The false statement is #1. A floating-point number is one that contains decimal positions.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Using Arithmetic Operators

Table 2-3 describes the five most commonly used **arithmetic operators**. You use these operators to manipulate values in your programs. The values that operators use in expressions are called **operands**. These arithmetic operators are called **binary operators** because you use two operands with each—one value to the left of the operator and another value to the right of it.

Operator	Description	Example
+	Addition	45 + 2: the result is 47
–	Subtraction	45 – 2: the result is 43
*	Multiplication	45 * 2: the result is 90
/	Division	45 / 2: the result is 22 (not 22.5)
%	Remainder (modulus)	45 % 2: the result is 1 (that is, 45 / 2 = 22 with a remainder of 1)

Table 2-3 Binary arithmetic operators

Unlike some other programming languages, C# does not have an exponential operator. Instead, you can use the built-in method `Math.Pow()` that you will learn about in the chapter "Introduction to Methods."

The operators / and % deserve special consideration. When you divide two numbers and at least one is a floating-point value, the answer is a floating-point value. For example, 45.0 / 2 is 22.5. However, when you divide integers using the / operator, whether they are integer constants or integer variables, the result is an integer; in other words, any fractional part of the result is lost. For example, the result of 45 / 2 is 22, not 22.5.

When you use the remainder operator with two integers, the result is an integer with the value of the remainder after division takes place—so the result of 45 % 2 is 1 because 2 "goes into" 45 twenty-two times with a remainder of 1. Remainder operations should not be performed with floating-point numbers. The results are unpredictable because of the way floating-point numbers are stored in computer memory.

In older languages, such as assembly languages, you had to perform division before you could take a remainder. In C#, performing a division operation is not necessary before using a remainder operation. In other words, a remainder operation can stand alone.

You may hear programmers refer to the remainder operator as the *modulus operator*. Although modulus has a similar meaning to remainder, there are differences.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Even though you define a result variable as a floating-point type, integer division still results in an integer. For example, the statement `double d = 7 / 2;` results in `d` holding 3, not 3.5, because the expression on the right is evaluated as integer-by-integer division before the assignment takes place. If you want the result to hold 3.5, at least one of the operands in the calculation must be a floating-point number, or else you must perform a cast. You will learn about casting later in this chapter.

When you combine mathematical operations in a single statement, you must understand **operator precedence**, or the rules that determine the order in which parts of a mathematical expression are evaluated. Multiplication, division, and remainder always take place prior to addition or subtraction in an expression. For example, the following expression results in 14:

```
int result = 2 + 3 * 4;
```

The result is 14 because the multiplication operation (3 * 4) occurs before adding 2. You can override normal operator precedence by putting the operation that should be performed first in parentheses. The following statement results in 20 because the addition within parentheses takes place first:

```
int result = (2 + 3) * 4;
```

In this statement, an intermediate result (5) is calculated before it is multiplied by 4.

Operator precedence is also called **order of operation**. A closely linked term is **associativity**, which specifies the order in which a sequence of operations with the same precedence are evaluated. Appendix A describes the precedence and associativity of every C# operator.

You can use parentheses in an arithmetic expression, even if they do not alter the default order of operation. For example, even though multiplication takes place before addition without parentheses, it is perfectly acceptable to add them as follows:

```
weeklyPay = (hours * rate) + bonus;
```

You can use parentheses to make your intentions clear to other programmers who read your programs, which means they do not have to rely on their memory of operator precedence.

In summary, the order of operations for arithmetic operators is:

- *Parentheses.* Evaluate expressions within parentheses first.
- *Multiplication, division, remainder.* Evaluate these operations next from left to right.
- *Addition, subtraction.* Evaluate these operations next from left to right.

Using Shortcut Arithmetic Operators

Increasing the value held in a variable is a common programming task. Assume that you have declared a variable named `counter` that counts the number of times an event has occurred. Each time the event occurs, you want to execute a statement such as the following:

```
counter = counter + 1;
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

This type of statement looks incorrect to an algebra student, but the equal sign (=) is not used to compare values in C#; it is used to assign values. The statement `counter = counter + 1;` says "Take the value of `counter`, add 1 to it, and assign the result to `counter`."

Because increasing the value of a variable is such a common task, C# provides several shortcut ways to count and accumulate. The following two statements are identical in meaning:

```
counter += 1;
counter = counter + 1;
```

The += operator is the **add and assign operator**; it adds the operand on the right to the operand on the left and assigns the result to the operand on the left in one step. For example, if `payRate` is 20, then `payRate += 5` results in `payRate` holding the value 25. Similarly, the following statements both increase `bankBal` by a rate stored in `interestRate`:

```
bankBal +=  bankBal * interestRate;
bankBal = bankBal + bankBal * interestRate;
```

For example, if `bankBal` is 100.00 and `interestRate` is 0.02, then both of the previous statements result in `bankBal` holding 102.00.

Additional shortcut operators in C# are -=,*=, and /=. Each of these operators is used to perform an operation and assign the result in one step. You cannot place spaces between the two characters that compose these operators; spaces preceding or following the operators are optional. Examples in which the shortcut operators are useful follow:

- `balanceDue -= payment` subtracts a payment from `balanceDue` and assigns the result to `balanceDue`.
- `rate *= 100` multiplies `rate` by 100 and assigns the result to `rate`. For example, it converts a fractional value stored in `rate`, such as 0.27, to a whole number, such as 27.
- `payment /= 12` changes a payment value from an annual amount to a monthly amount due.

When you want to increase a variable's value by exactly 1, you can use either of two other shortcut operators—the **prefix increment operator** and the **postfix increment operator**. To use a prefix increment operator, you type two plus signs before the variable name. For example, these statements result in `someValue` holding 7:

```
int someValue = 6;
++someValue;
```

The variable `someValue` holds 1 more than it held before the ++ operator was applied. To use a postfix ++, you type two plus signs just after a variable name. Executing the following statements results in `anotherValue` holding 57:

```
int anotherValue = 56;
anotherValue++;
```

You can use the prefix ++ and postfix ++ with variables but not with constants. An expression such as ++84 is illegal because 84 is constant and must always remain 84. However, you can create a variable as in `int val = 84;`, and then write either `++val` or `val++` to increase the variable's value to 85.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The prefix and postfix increment operators are **unary operators** because you use them with one operand. Most arithmetic operators, like those used for addition and multiplication, are binary operators that operate on two operands.

When all you want to accomplish is to increase a variable's value by 1, there is no apparent difference between using the prefix and postfix increment operators. However, these operators function differently. When you use the prefix ++, the result is calculated and stored and then the variable is used. For example, in the following code, both `b` and `c` end up holding 5. The `WriteLine()` statement displays "5 5". In this example, 4 is assigned to `b`, then `b` becomes 5, and then 5 is assigned to `c`.

```
b = 4;
c = ++b;
WriteLine("{0} {1}", b, c);
```

In contrast, when you use the postfix ++, the variable is used, and then the result is calculated and stored. For example, in the second line of the following code, 4 is assigned to `c`; then, *after* the assignment, `b` is increased and takes the value 5.

```
b = 4;
c = b++;
WriteLine("{0} {1}", b, c);
```

This last `WriteLine()` statement displays "5 4". In other words, if `b = 4`, then the value of `b++` is also 4, and that value is assigned to `c`. However, after the 4 is assigned to `c`, `b` is increased to 5.

When you need to add 1 to a variable in a stand-alone statement, the results are the same whether you use a prefix or postfix increment operator. However, many programmers routinely use the postfix operator when they could use either operator. This is probably a mistake because the prefix operator is more efficient. You will see an example that proves the superior efficiency of the prefix operator in the chapter "Looping."

A prefix or postfix **decrement operator** (--) reduces a variable's value by 1. For example, if `s` and `t` are both assigned the value 34, then the expression `--s` has the value 33 and the expression `t--` has the value 34, but `t` then becomes 33.

Watch the video *Using Shortcut Arithmetic Operators*.

TWO TRUTHS & A LIE

Using Arithmetic Operators

1. The value of 26 % 4 * 3 is 18.
2. The value of 4 / 3 + 2 is 3.
3. If `price` is 4 and `tax` is 5, then the value of `price - tax++` is –1.

The false statement is #1. The value of 26 % 4 * 3 is 6. The value of the first part of the expression, 26 % 4, is 2, because 2 is the remainder when 4 is divided into 26. Then 2 * 3 is 6.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You Do It

Performing Arithmetic

In the following steps, you add some arithmetic statements to a program.

1. Open a new C# program file named **DemoVariables2**, and enter the following statements to start a program that demonstrates arithmetic operations:

```
using static System.Console;
class DemoVariables2
{
   static void Main()
   {
```

2. Write a statement that declares seven integer variables. Assign initial values to two of the variables; the values for the other five variables will be calculated. Because all of these variables are the same type, you can use a single statement to declare all seven integers. Recall that to do this, you insert commas between variable names and place a single semicolon at the end. You can place line breaks wherever you want for readability. (Alternatively, you could use as many as seven separate declarations.)

```
int value1 = 43, value2 = 10,
   sum, diff, product, quotient, remainder;
```

3. Write the arithmetic statements that calculate the sum of, difference between, product of, quotient of, and remainder of the two assigned variables.

```
sum = value1 + value2;
diff = value1 - value2;
product = value1 * value2;
quotient = value1 / value2;
remainder = value1 % value2;
```

Instead of declaring the variables `sum`, `diff`, `product`, `quotient`, and `remainder` and assigning values later, you could declare and assign all of them at once, as in `int sum = value1 + value2;`. The only requirement is that `value1` and `value2` must be assigned values before you can use them in a calculation.

4. Include five `WriteLine()` statements to display the results.

```
WriteLine("The sum of {0} and {1} is {2}",
   value1, value2, sum);
WriteLine("The difference between {0} and {1}" +
   "is {2}", value1, value2, diff);
WriteLine("The product of {0} and {1} is {2}",
   value1, value2, product);
```

(continues)

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

```
WriteLine("{0} divided by {1} is {2}", value1,
   value2, quotient);
WriteLine("and the remainder is {0}", remainder);
```

5. Add two closing curly braces—one for the `Main()` method and the other for the `DemoVariables2` class.
6. Save the file, compile the program, and execute it. The output should look like Figure 2-11.

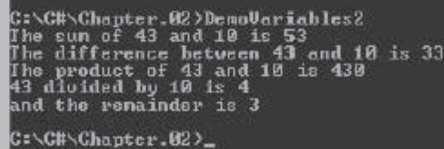

Figure 2-11 Output of the `DemoVariables2` program

7. Change the values of the `value1` and `value2` variables, save the program, and compile and run it again. Repeat this process several times. After each execution, analyze the output to make sure you understand the results of the arithmetic operations.

Using the bool Data Type

Boolean logic is based on true-or-false comparisons. An `int` variable can hold millions of different values at different times, but a **Boolean variable** can hold only one of two values—true or false. You declare a Boolean variable by using type **`bool`**. The following statements declare and assign appropriate values to two `bool` variables:

```
bool isItMonday = false;
bool areYouTired = true;
```

If you begin a `bool` variable name with a form of the verb "to be" or "to do," such as "is" or "are," then you can more easily recognize the identifiers as Boolean variables when you encounter them within your programs.

When you use "Boolean" as an adjective, as in "Boolean variable," you usually begin with an uppercase *B* because the data type is named for Sir George Boole, the founder of symbolic logic, who lived from 1815 to 1864. The C# data type `bool`, however, begins with a lowercase *b*.

You also can assign values based on the result of comparisons to Boolean variables. A **comparison operator** compares two items; an expression containing a comparison operator has a Boolean value. Table 2-4 describes the six comparison operators that C# supports.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Operator	Description	true Example	false Example
<	Less than	3 < 8	8 < 3
>	Greater than	4 > 2	2 > 4
==	Equal to	7 == 7	3 == 9
<=	Less than or equal to	5 <=5	8 <= 6
>=	Greater than or equal to	7 >= 3	1 >= 2
!=	Not equal to	5 != 6	3 != 3

Table 2-4 Comparison operators

When using any of the operators that require two keystrokes (==, <=, >=, or !=), you cannot place any whitespace between the two symbols.

A frequent mistake among new programmers is attempting to create a Boolean expression for equivalency with a single equal sign. The equivalency operator uses two equal signs. For example, the following assigns either `true` or `false` to `isDiscountProvided`:

```
isDiscountProvided = custStatus == 1;
```

The second operator in the statement (==) compares `custStatus` to 1. The result is then assigned to the variable on the left using the assignment operator (=).

Legal (but somewhat useless) declaration statements might include the following, which compare two values directly:

```
bool isBigger = 6 > 5; // Value stored would be true
bool isSmallerOrEqual = 7 <= 4; // Value stored would be false
```

Using Boolean values is more meaningful when you use variables (that have been assigned values) rather than constants in the comparisons, as in the following examples:

```
bool doesEmployeeReceiveOvertime = hoursWorked > 40;
bool isHighTaxBracket = annualIncome > 100000;
```

In the first statement, the `hoursWorked` variable is compared to a constant value of 40. If the `hoursWorked` variable holds a value less than or equal to 40, then the expression is evaluated as `false`. In the second statement, the `annualIncome` variable value must be greater than 100000 for the expression to be `true`.

Boolean variables become more useful after you learn to make decisions within C# programs in Chapter 4.

When you display a `bool` variable's value with a call to `Write()` or `WriteLine()`, the displayed value is capitalized as `True` or `False`. However, the values within programs are lowercase: `true` or `false`.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

TWO TRUTHS & A LIE

Using the bool Data Type

1. If `rate` is 7.5 and `min` is 7, then the value of `rate >= min` is `false`.
2. If `rate` is 7.5 and `min` is 7, then the value of `rate < min` is `false`.
3. If `rate` is 7.5 and `min` is 7, then the value of `rate == min` is `false`.

The false statement is #1. If `rate` is 7.5 and `min` is 7, then the value of `rate >= min` is `true`.

You Do It

Working with Boolean Variables

Next, you write a program that demonstrates how Boolean variables operate.

1. Open a new file in your editor, and name it **DemoVariables3.**
2. Enter the following code. In the `Main()` method, you declare an integer value, then assign different values to a Boolean variable. Notice that when you declare `value` and `isSixMore`, you assign types. When you reassign values to these variables later in the program, you do not redeclare them by using a type name. Instead, you simply assign new values to the already declared variables.

```
using static System.Console;
class DemoVariables3
{
   static void Main()
   {
      int value = 4;
      bool isSixMore = 6 > value;
      WriteLine("When value is {0} isSixMore is {1}",
         value, isSixMore);
      value = 35;
      isSixMore = 6 > value;
      WriteLine("When value is {0} isSixMore is {1}",
         value, isSixMore);
   }
}
```

3. Save, compile, and run the program. The output looks like Figure 2-12.

(continues)

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

```
C:\C#\Chapter.02>DemoVariables3
When value is 4 isSixMore is True
When value is 35 isSixMore is False

C:\C#\Chapter.02>
```

Figure 2-12 Output of the `DemoVariables3` program

4. Change the value of the variable named `value`, and try to predict the outcome. Run the program to confirm your prediction.

Understanding Numeric Type Conversion

When you perform arithmetic with variables or constants of the same type, the result of the arithmetic retains that type. For example, when you divide two `int`s, the result is an `int`; when you subtract two `double`s, the result is a `double`. Often, however, you need to perform mathematical operations on different types. For example, in the following code, you multiply an `int` by a `double`:

```
int hoursWorked = 36;
double payRate = 12.35;
double grossPay = hoursWorked * payRate;
```

When you perform arithmetic operations with operands of dissimilar types, C# chooses a **unifying type** for the result and **implicitly** (or automatically) converts nonconforming operands to the unifying type, which is the type with the higher **type precedence**. The conversion is called an **implicit conversion** or an **implicit cast**—the automatic transformation that occurs when a value is assigned to a type with higher precedence.

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `short` to `int`, `long`, `float`, `double`, or `decimal`
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `int` to `long`, `float`, `double`, or `decimal`
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`
- From `long` to `float`, `double`, or `decimal`
- From `ulong` to `float`, `double`, or `decimal`
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
- From `float` to `double`

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Implicit conversions occur in simple assignments as well as in calculations. For example, if `money` is declared as a `double`, then the following statement implicitly converts the integer 15 to a `double`:

```
money = 15;
```

Conversions from `int`, `uint`, or `long` to `float` and from `long` to `double` may cause a loss of precision but will never cause a loss of magnitude.

A constant expression of type `int`, such as 25, can be converted to `sbyte, byte, short, ushort, uint`, or `ulong`. For example, `sbyte age = 19;`, which assigns an `int` to an `sbyte`, is legal. However, you must make sure that the assigned value is within the allowed range for the destination type, or the program will not compile.

For example, if you multiply an `int` and a `double`, the result is implicitly a `double`. This requirement means the result must be stored in a `double`; if you attempt to assign the result to an `int`, you will receive a compiler error message like the one shown in Figure 2-13.

```
TryingToStoreDoubleInInt.cs(8,17): error CS0266: Cannot implicitly convert type
        'double' to 'int'. An explicit conversion exists (are you missing a
        cast?)

C:\C#\Chapter.02>
```

Figure 2-13 Error message received when trying to compile a program that attempts to store a `double` in an `int`

The error message in Figure 2-13 asks "are you missing a cast?" You may **explicitly** (or purposefully) override the unifying type imposed by C# by performing an explicit cast. An **explicit cast** involves placing the desired result type in parentheses followed by the variable or constant to be cast. For example, two explicit casts are performed in the following code:

```
double bankBalance = 191.66;
float weeklyBudget = (float) bankBalance / 4;
   // weeklyBudget is 47.915, one-fourth of bankBalance
int dollars = (int) weeklyBudget;
   // dollars is 47, the integer part of weeklyBudget
```

The value of `bankBalance / 4` is implicitly a `double` because a `double` divided by an `int` produces a `double`. The explicit cast then converts the `double` result to a `float` before it is stored in `weeklyBudget`. Similarly, the `float` value `weeklyBudget` is cast to an `int` before it is stored in `dollars` (and the decimal-place values are lost).

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

It is easy to lose data when performing a cast. For example, the largest `byte` value is 255, and the largest `int` value is 2,147,483,647, so the following statements produce distorted results because some information is lost during the cast:

- `int anOkayInt = 256;`
- `byte aBadByte = (byte)anOkayInt;`

In this example, displaying `aBadByte` shows 0. If you stored 257 in `anOkayInt`, the result would appear as 1; if you stored 258, the result would appear as 2; and so on.

Watch the video *Numeric Type Conversion.*

TWO TRUTHS & A LIE

Understanding Numeric Type Conversion

1. If `deptNum` is an `int` with a value of 10, then the following is valid:
   `double answer = deptNum;`
2. If `answer` is a `double` with a value of 10.0, then the following is valid:
   `int deptNum = answer;`
3. If `value` is a `double` with a value of 12.2, then the following is valid:
   `double answer = (int)value;`

The false statement is #2. A `double` cannot be implicitly converted to an `int`.

Using the `char` Data Type

You use the **`char`** data type to hold any single character. You place constant character values within single quotation marks because the computer stores characters and integers differently. For example, the following statement assigns *D* to initial:

```
char initial = 'D';
```

A number can be a character, in which case it must be enclosed in single quotation marks and declared as a `char` type, as in the following:

```
char aCharValue = '9';
```

When you store a digit such as `'9'` as a character, you cannot use it in ordinary arithmetic statements. If the value of 9 is needed for use in arithmetic, you should store it without quotes as a numeric type, such as `int`.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

A variable of type `char` can hold only one character, and a literal character is written using single quotation marks. To store a string of characters, such as a person's name, you must use a `string`; a literal string is written using double quotation marks. You will learn about `string`s later in this chapter.

You can store any character—including nonprinting characters such as a backspace or a tab—in a `char` variable. To store these characters, you use two symbols in an **escape sequence**, which always begins with a backslash. The pair of symbols represents a single character. For example, the following code stores a backspace character and a tab character in the `char` variables `aBackspaceChar` and `aTabChar`, respectively:

```
char aBackspaceChar = '\b';
char aTabChar = '\t';
```

In the preceding code, the escape sequence indicates a unique value for each character—a backspace or tab instead of the letter *b* or *t*. Table 2-5 describes some common escape sequences that are used in C#.

Escape Sequence	Character Name
`\'`	Single quotation mark
`\"`	Double quotation mark
`\\`	Backslash
`\0`	Null
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

Table 2-5 Common escape sequences

The characters used in C# are represented in Unicode, which is a 16-bit coding scheme for characters. For example, the letter *A* actually is stored in computer memory as a set of 16 zeros and ones—namely, 0000 0000 0100 0001. (The spaces are inserted here after every set of four digits for readability.) Because 16-bit numbers are difficult to read, programmers often use a shorthand notation called **hexadecimal**, or **base 16**. In hexadecimal shorthand, 0000 becomes 0,

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

0100 becomes 4, and 0001 becomes 1. Thus, the letter *A* is represented in hexadecimal as 0041. You tell the compiler to treat the four-digit hexadecimal 0041 as a single character by preceding it with the `\u` escape sequence. Therefore, there are two ways to store the character *A*:

```
char letter = 'A';
char letter = '\u0041';
```

For more information about Unicode, see Appendix B.

The second option, using hexadecimal, obviously is more difficult and confusing than the first option, so it is not recommended that you store letters of the alphabet using the hexadecimal method. However, you can produce some interesting values using the Unicode format. For example, letters from foreign alphabets that are not on a standard keyboard (such as letters from Greek, Hebrew, and Chinese alphabets) and other special symbols (foreign currency symbols, mathematical symbols, geometric shapes, and so on) are available in Unicode. You can even output the sound of a beep by displaying the hexadecimal character `'\u0007'`.

TWO TRUTHS & A LIE

Using the char Data Type

1. The following statement is legal in C#:

   `char department = '5';`

2. The following statement is legal in C#:

   `char department = '\f';`

3. The following statement is legal in C#:

   `char department = '32';`

The false statement is #3. Only a single character can appear between single quotation marks, with the exception of escape sequence characters such as `'\t'` and `'\f'`. In these cases, a single character is created using two symbols.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You Do It

Using Escape Sequences

Next, you write a short program to demonstrate the use of escape sequences.

1. Open a new file, and name it **DemoEscapeSequences.**
2. Enter the following code. The three `WriteLine()` statements demonstrate using escape sequences for tabs, a newline, and alerts.

```
using static System.Console;
class DemoEscapeSequences
{
   static void Main()
   {
      WriteLine("This line\tcontains two\ttabs");
      WriteLine("This statement\ncontains a new line");
      WriteLine("This statement sounds three alerts\a\a\a");
   }
}
```

3. Save, compile, and execute the program. Your output should look like Figure 2-14. Additionally, if your system has speakers and they are on, you should hear three "beep" sounds caused by the three alert characters: '\a'.

```
C:\C#\Chapter.02>DemoEscapeSequences
This line       contains two    tabs
This statement
contains a new line
This statement sounds three alerts

C:\C#\Chapter.02>
```

Figure 2-14 Output of the `DemoEscapeSequences` program

Using the `string` Data Type

In C#, you use the **`string`** data type to hold a series of characters. The value of a `string` is expressed within double quotation marks. For example, the following statement declares a `string` named `firstName` and assigns *Jane* to it:

```
string firstName = "Jane";
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

You can also initialize a string with a character array. The “Using Arrays” chapter explains more.

You can compare the value of one `string` to another using the == and != operators in the same ways that you compare numeric or character variables. For example, the program in Figure 2-15 declares three `string` variables. Figure 2-16 shows the results: `string`s that contain *Amy* and *Amy* are considered equal, but `strings` that contain *Amy* and *Matthew* are not.

```
using static System.Console;
class CompareNames1
{
   static void Main()
   {
      string name1 = "Amy";
      string name2 = "Amy";
      string name3 = "Matthew";
      WriteLine("compare {0} to {1}: {2}",
         name1, name2, name1 == name2);
      WriteLine("compare {0} to {1}: {2}",
         name1, name3, name1 == name3);
   }
}
```

Figure 2-15 Program that compares two strings using == operator (not recommended)

```
C:\C#\Chapter.02>CompareNames1
compare Amy to Amy: True
compare Amy to Matthew: False

C:\C#\Chapter.02>_
```

Figure 2-16 Output of the `CompareNames1` program

In addition to the == comparison operator, you can use several prewritten methods to compare `string`s. The advantage to using these other methods is that they have standard names; when you use most other C# classes written by others, they will use methods with the same names to compare their objects. You can compare `string`s with any of the following methods:

- The `String` class `Equals()` method requires two `string` arguments that you place within its parentheses, separated by a comma. As when you use the == operator, the `Equals()` method returns `true` or `false`.
- The `String` class `Compare()` method also requires two `string` arguments, but it returns an integer. When it returns 0, the two `string`s are equivalent; when it returns a positive

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

number, the first `string` is greater than the second; and when it returns a negative value, the first `string` is less than the second. A `string` is considered equal to, greater than, or less than another string **lexically**, which in the case of letter values means alphabetically. That is, when you compare two `strings`, you compare each character in turn from left to right. If each Unicode value is the same, then the strings are equivalent. If any corresponding character values are different, the `string` that has the greater Unicode value earlier in the string is considered greater.

- The `CompareTo()` method also compares two `strings`. However, one `string` is used before the dot and method name, and the second `string` is placed within the method's parentheses as an argument. Like the `Compare()` method, the `CompareTo()` method returns a 0 when the compared `strings` are equal, a negative number if the first `string` is less, and a positive number if the second `string` (the one in parentheses) is less.

Figure 2-17 shows a program that makes several comparisons using the three methods; in each case the method name is shaded. Figure 2-18 shows the program's output.

Methods like `Compare()` that are not preceded with an object and a dot when they are called are *static methods*. Methods like `CompareTo()` that are preceded with an object and a dot are nonstatic and are *instance methods*. The "Advanced Method Concepts" chapter explains more.

```
using System;
using static System.Console;
class CompareTwoNames
{
   static void Main()
   {
      string name1 = "Amy";
      string name2 = "Amy";
      string name3 = "Matthew";
      WriteLine("Using Equals() method");
      WriteLine("compare {0} to {1}: {2}",
         name1, name2, String.Equals(name1, name2));
      WriteLine("compare {0} to {1}: {2}",
         name1, name3, String.Equals(name1, name3));
      WriteLine("Using Compare() method");
      WriteLine(" compare {0} to {1}: {2}",
         name1, name2, String.Compare(name1, name2));
      WriteLine("compare {0} to {1}: {2}",
         name1, name3, String.Compare(name1, name3));
      WriteLine("Using CompareTo() method");
      WriteLine("compare {0} to {1}: {2}",
         name1, name2, name1.CompareTo(name2));
      WriteLine("compare {0} to {1}: {2}",
         name1, name3, name1.CompareTo(name3));
   }
}
```

Figure 2-17 Program that compares two strings using three methods

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
C:\C#\Chapter.02>CompareTwoNames
Using Equals() method
   compare Amy to Amy: True
   compare Amy to Matthew: False
Using Compare() method
   compare Amy to Amy: 0
   compare Amy to Matthew: -1
Using CompareTo() method
   compare Amy to Amy: 0
   compare Amy to Matthew: -1

C:\C#\Chapter.02>_
```

Figure 2-18 Output of the `CompareTwoNames` program

In C#, a string is **immutable**. That is, a string's value is not actually modified when you assign a new value to it. For example, when you write `name = "Amy";` followed by `name = "Donald";`, the first literal string of characters, *Amy*, still exists in computer memory, but the `name` variable no longer refers to that string's memory address. Instead, `name` refers to a new address where the characters in *Donald* are stored. The situation is different than with numbers; when you assign a new value to a numeric variable, the value at the named memory address actually changes, erasing the original value.

The comparisons made by the `Equals()`, `Compare()`, and `CompareTo()` methods are case sensitive. In other words, "Amy" does not equal "amy". In Unicode, the decimal value of each uppercase letter is exactly 32 less than its lowercase equivalent. For example, the decimal value of a Unicode 'a' is 97 and the value of 'A' is 65.

You can use the `Length` property of a string to determine its length. For example, suppose you declare a string as follows:

```
string word = "water";
```

The value of `word.Length` is 5. Notice that `Length` is not a method and that it is not followed with parentheses.

The `Substring()` method can be used to extract a portion of a string from a starting point for a specific length. For example, Figure 2-19 shows a string that contains the word *water*. The first character in a string is at position 0, and the last character is at position `Length – 1`. Therefore, after you declare `string word = "water";`, each of the first five expressions in the figure is true. The last expression produces an error message because the combination of the starting position and length attempt to access values outside the range of the string.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

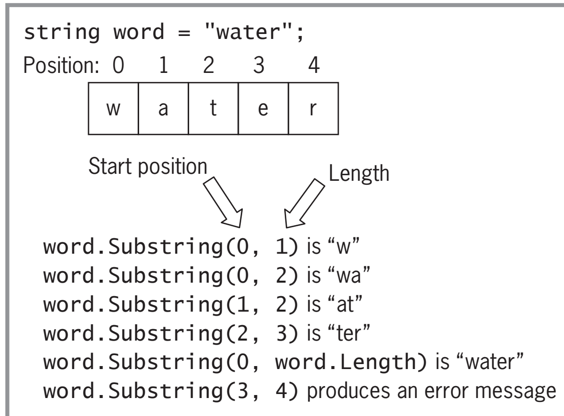

Figure 2-19 Using the `Substring()` method

The `StartsWith()` method determines whether a string starts with the characters in the string that is used as an argument. For example, if `word` is "water", the expression `word.StartsWith("wa")` is true, and the expression `word.StartsWith("waet")` is false.

TWO TRUTHS & A LIE

Using the `string` Data Type

1. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Equals(creature1, creature2)` is false.
2. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Compare(creature1, creature2)` is false.
3. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `creature1.CompareTo(creature2)` is a positive number.

The false statement is #2. If `creature1 = "dog"` and `creature2 = "cat"`, then the value of `String.Compare(creature1, creature2)` is a positive number.

Defining Named Constants

By definition, a variable's value can vary or change. Sometimes you want to create a **named constant**, an identifier for a memory location whose contents cannot change. You create a named constant by adding the keyword `const` before the data type in a declaration. Although

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

there is no requirement to do so, programmers usually name constants using all uppercase letters, inserting underscores for readability. This convention makes constant names stand out so that a reader is less likely to confuse them with variable names. For example, the following declares a constant named TAX_RATE that is initialized to 0.06:

```
const double TAX_RATE = 0.06;
```


You must assign a value to a constant when you create it. You can use a constant just as you would use a variable of the same type—for example, display it or use it in a mathematical equation—but you cannot assign any new value to it.

It's good programming practice to use named constants for any values that should never change; doing so makes your programs clearer. For example, when you declare a constant `const int INCHES_IN_A_FOOT = 12;` within a program, then you can use a statement such as the following:

```
lengthInInches = lengthInFeet * INCHES_IN_A_FOOT;
```

This statement is **self-documenting**; that is, even without a program comment, it is easy for someone reading your program to tell why you performed the calculation in the way you did.

Working with Enumerations

An **enumeration** is a set of constants represented by identifiers. You place an enumeration in a class outside of any methods. Very frequently, programmers place enumerations at the top of a file before any methods. You create an enumeration using the keyword `enum` and an identifier, followed by a comma-separated list of constants between curly braces. By convention, the enumerator identifier begins with an uppercase letter. For example, the following is an enumeration called `DayOfWeek`:

```
enum DayOfWeek
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
      THURSDAY, FRIDAY, SATURDAY
}
```

By default, enumeration values are integers. You could also declare an `enum`'s type to be `byte`, `sbyte`, `short`, `ushort`, `uint`, `long`, or `ulong` by including a colon and the type after the enumeration name and before the opening curly brace of the assignment list.

Frequently, the identifiers in an enumeration are meant to hold consecutive values. When you do not supply values for the items in an `enum` list, they start with 0 and are automatically increased by 1. For example, in the `DayOfWeek` enumeration, `SUNDAY` is 0, `MONDAY` is 1, and so on.

As is the convention with other constants, you might prefer to name `enum` constants using all uppercase letters and underscores for readability, as shown in the `DayOfWeek` enumeration here. This helps distinguish enumeration names from variables. However, the developers of C# frequently violate this convention in both their online documentation and in built-in enumerations in the language. For example, the built-in `FontStyle` enumeration contains constants with mixed-case names such as `Bold` and `Italic`. When you create enumerations, you should follow the convention your instructor or supervisor prefers. Another convention is to use a singular noun or noun phrase as the `enum` identifier. In other words, `DayOfWeek` is better than `DaysOfWeek`.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

If you want the first `enum` constant in the list to have a value other than 0, you can assign one, as in the following:

```
enum DayOfWeek
{
   SUNDAY = 1, MONDAY, TUESDAY, WEDNESDAY,
     THURSDAY, FRIDAY, SATURDAY
}
```

In this case, `SUNDAY` is 1, `MONDAY` is 2, `TUESDAY` is 3, and so on.

As another option, you can assign values to each enumeration, as in the following:

```
enum SpeedLimit
{
   SCHOOL_ZONE_SPEED = 20,
   CITY_STREET_SPEED = 30,
   HIGHWAY_SPEED = 55
}
```

The names used within an `enum` must be unique, but the values assigned don't have to be.

When you create an enumeration, you clearly identify what values are valid for an item. You also make your code more self-documenting. You should consider creating an enumeration any time you have a fixed number of integral values allowed for a variable.

As a side benefit, when you use an `enum` in Visual Studio, Intellisense lists the defined values. Appendix C provides more information on Intellisense.

After you have created an enumeration, you can declare variables of the enumeration type, and you can assign enumeration values to them. For example, you can use the following statements to declare `payrollDay` as data type `DayOfWeek`, and assign a value to it:

```
DayOfWeek payrollDay;
payrollDay = DayOfWeek.TUESDAY;
```

In the chapter "Using Classes and Objects," you will learn to create many additional data types that are more complex than the built-in data types.

You can convert an `enum` value to another data type or an integral type to an `enum` value by using a cast. For example, the following are valid statements:

```
int shipDay;
DayOfWeek deliverDay;
shipDay = (int)DayOfWeek.THURSDAY;
deliverDay = (DayOfWeek)shipDay;
```

Using the preceding example, you can display *5* and *THURSDAY* with the following statements:

```
WriteLine(shipDay);
WriteLine(deliverDay);
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Creating an enumeration type provides you with several advantages. For example, the `DayOfWeek` enumeration improves your programs in the following ways:

- If you did not create an enumerated type, you could use another type—for example, `int`s or `string`s. The problem is that any value could be assigned to an `int` or `string` variable, but only the seven allowed values can be assigned to a `DayOfWeek` object.
- If you did not create an enumerated type, you could create another type to represent days, but invalid behavior could be applied to the values. For example, if you used integers to represent days, you could multiply or divide two days, which is not logical. Programmers say using `enum`s makes the values type-safe. **Type-safe** describes a data type for which only appropriate behaviors are allowed.
- The `enum` constants provide a form of self-documentation. Someone reading your program might misinterpret what 3 means as a day value, but there is less confusion when you use the identifier `WEDNESDAY`.

This discussion is meant to be a brief introduction to enumerations. At this point in your study of C#, the advantages to using enumerations are limited. Enumerations are particularly useful with `switch` statements, which you will learn about in the "Making Decisions" chapter.

TWO TRUTHS & A LIE

Defining Named Constants

1. The following is a valid and conventional C# constant declaration:

```
const string FIRST_HEADING = "Progress Report";
```

2. The following is a valid and conventional C# constant declaration:

```
const double COMMISSION_RATE;
```

3. By default, enumeration values are integers.

The false statement is #2. A constant must be assigned a value when it is declared.

Accepting Console Input

A program that allows user input is an **interactive program**. You can use the `ReadLine()` method to create an interactive program that accepts user input from the keyboard. This method accepts all of the characters entered by a user until the user presses Enter.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The characters can be assigned to a `string`. For example, the following statement accepts a user's input and stores it in the variable `myString`:

```
myString = ReadLine();
```


The `Read()` method is similar to the `ReadLine()` method. `Read()` reads just one character from the input stream, whereas `ReadLine()` reads every character in the input stream until the user presses Enter.

Prior to the release of C# 6.0, the qualified names `Console.ReadLine()` and `Console.Read()` were required. However, just like with `WriteLine()` and `Write()`, you can now include `using static System.Console;` at the top of a file allowing you to use the shorter method names `ReadLine()` and `Read()`.

If you want to use the input value as a type other than `string`, then you must use a conversion method to change the input `string` to the proper type. Two methods are available for converting strings—the `Convert` class and `Parse()` methods.

Using the `Convert` Class

Figure 2-20 shows an interactive program that prompts the user for a price and calculates a 6 percent sales tax. The program displays *Enter the price of an item* on the screen. Such an instruction to the user is a **prompt**. Notice that two using statements appear at the top of the file:

- `using System;` to include the `Convert` class
- `using static System.Console;` to include the `ReadLine()` and `WriteLine()` methods

Notice that the program uses a `Write()` statement for the prompt instead of a `WriteLine`() statement so that the user's input appears on the screen on the same line as the prompt. Also notice that the prompt contains a space just before the closing quote so that the prompt does not crowd the user's input on the screen.

After the prompt appears, the `ReadLine()` statement accepts a string of characters and assigns them to the variable `itemPriceAsString`. Before the tax can be calculated, this value must be converted to a number. This conversion is accomplished using the `Convert` class `ToDouble()` method in the shaded statement. Figure 2-21 shows a typical execution of the program in which the user typed 28.77 as the input value.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
using System;
using static System.Console;
class InteractiveSalesTax
{
   static void Main()
   {
      const double TAX_RATE = 0.06;
      string itemPriceAsString;
      double itemPrice;
      double total;
      Write("Enter the price of an item >> ");
      itemPriceAsString = ReadLine();
      itemPrice = Convert.ToDouble(itemPriceAsString);
      total = itemPrice * TAX_RATE;
      WriteLine("With a tax rate of {0}, a {1} item " +
         "costs {2} more.", TAX_RATE, itemPrice.ToString("C”),
         total.ToString("C"));
   }
}
```

Figure 2-20 The `InteractiveSalesTax` program

In the program in Figure 2-20, the angle brackets at the end of the prompt are not required; they are merely cosmetic. You might prefer other punctuation such as a colon, ellipsis, or hyphen to indicate that input is required.

```
C:\C#\Chapter.02>InteractiveSalesTax
Enter the price of an item >> 28.77
With a tax rate of 0.06, a $28.77 item costs $1.73 more.

C:\C#\Chapter.02>
```

Figure 2-21 Typical execution of the `InteractiveSalesTax` program

As a shortcut, you can avoid declaring and using the extra `string` variable `itemPriceAsString` in the program in Figure 2-20. Instead, you can accept and convert the input string in one step by nesting the `ReadLine()` and `ToDouble()` method calls, as in the following:

```
itemPrice = Convert.ToDouble(ReadLine());
```

Table 2-6 shows `Convert` class methods you can use to change `string`s into additional useful data types. The methods use the system types (also called runtime types) in their names. For example, recall from Table 2-1 that the “formal” name for an `int` is `Int32`, so the method you use to convert a `string` to an `int` is named `Convert.ToInt32()`.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Method	Description
`ToBoolean()`	Converts a specified value to an equivalent Boolean value
`ToByte()`	Converts a specified value to an 8-bit unsigned integer
`ToChar()`	Converts a specified value to a Unicode character
`ToDecimal()`	Converts a specified value to a decimal number
`ToDouble()`	Converts a specified value to a double-precision floating-point number
`ToInt16()`	Converts a specified value to a 16-bit signed integer (a `short`)
`ToInt32()`	Converts a specified value to a 32-bit signed integer (an `int`)
`ToInt64()`	Converts a specified value to a 64-bit signed integer (a `long`)
`ToSByte()`	Converts a specified value to an 8-bit signed integer
`ToSingle()`	Converts a specified value to a single-precision floating-point number
`ToString()`	Converts the specified value to its equivalent `String` representation
`ToUInt16()`	Converts a specified value to a 16-bit unsigned integer
`ToUInt32()`	Converts a specified value to a 32-bit unsigned integer
`ToUInt64()`	Converts a specified value to a 64-bit unsigned integer

Table 2-6 Selected `Convert` class methods

Using the `Parse()` Methods

As an alternative to the `Convert` class methods, you can use a `Parse()` method to convert a string to a number. For example, to convert a `string` to a `double` in the `InteractiveSalesTax` program in Figure 2-20, you can replace the `Convert.ToDouble()` statement with either of the following `Parse()` method statements:

```
itemPrice = double.Parse(itemPriceAsString);
itemPrice = Double.Parse(itemPriceAsString);
```

Each of the previous two statements uses a `Parse()` method defined in the `Double` class. You can use the system type name `Double` or its simple type name `double`. Either version converts the `string` argument within the parentheses to the correct data type.

Similar methods exist for other data types, such as `int.Parse()` and `boolean.Parse()`. For example, either of the following two statements reads an integer and stores it in a variable named `score` without using a temporary string to hold the input value:

```
score = int.Parse(ReadLine());
score = Int32.Parse(ReadLine());
```

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

There are some performance differences between the `Convert` and `Parse()` techniques, but they are minor and relatively technical. Until you learn more about C#, you should use the method preferred by your instructor or organization.

The term **parse** means to break into component parts. Grammarians talk about "parsing a sentence"—deconstructing it so as to describe its grammatical components. Parsing a `string` converts it to its numeric equivalent.

TWO TRUTHS & A LIE

Accepting Console Input

1. The following is valid:

```
int age = Convert.ToInt(ReadLine());
```

2. The following is valid:

```
double payRate = Convert.ToDouble(ReadLine());
```

3. The following is valid:

```
int dependents = int.Parse(ReadLine());
```

The false statement is #1. The `Convert` class method to convert a string to an integer is `ToInt32()`, not `ToInt()`.

You Do It

Writing a Program that Accepts User Input

In the next steps, you write an interactive program that allows the user to enter two integer values. The program then calculates and displays their sum.

1. Open a new file named **InteractiveAddition**, and type the first few lines needed for the `Main()` method:

```
using System;
using static System.Console;
class InteractiveAddition
{
   static void Main()
   {
```

(continues)

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

(continued)

2. Add variable declarations for two `strings` that will accept the user's input values. Also, declare three integers for the numeric equivalents of the `string` input values and their sum.

```
string name, firstString, secondString;
int first, second, sum;
```

3. Prompt the user for his or her name, accept it into the `name string`, and then display a personalized greeting to the user, along with the prompt for the first integer value.

```
Write("Enter your name... ");
name = ReadLine();
Write("Hello {0}! Enter the first integer... ", name);
```

4. Accept the user's input as a `string`, and then convert the input `string` to an integer.

```
firstString = ReadLine();
first = Convert.ToInt32(firstString);
```

5. Add statements that prompt for and accept the second `string` and convert it to an integer.

```
Write("Enter the second integer... ");
secondString = ReadLine();
second = Convert.ToInt32(secondString);
```

6. Assign the sum of the two integers to the `sum` variable and display all of the values. Add the closing curly brace for the `Main()` method and the closing curly brace for the class.

```
      sum = first + second;
      WriteLine("{0}, the sum of {1} and {2} is {3}",
         name, first, second, sum);
   }
}
```

7. Save, compile, and run the program. When prompted, supply your name and any integers you want, and confirm that the result appears correctly. Figure 2-22 shows a typical run of the program.

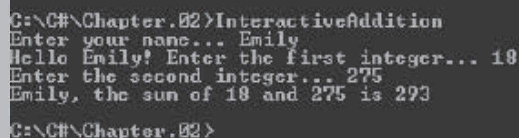

Figure 2-22 Typical execution of the `InteractiveAddition` program

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Chapter Summary

- A constant is a data value that cannot be changed after a program is compiled; a value in a variable can change. C# provides for 15 basic built-in types of data. A variable declaration includes a data type, an identifier, an optional assigned value, and a semicolon.
- You can display variable values by using the variable name within a `WriteLine()` or `Write()` method call. To make producing output easier, you can combine strings and variable values into a single `Write()` or `WriteLine()` statement by using a format string.
- C# supports nine integral data types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`.
- C# supports three floating-point data types: `float`, `double`, and `decimal`. You can use format and precision specifiers to display floating-point data to a specified number of decimal places.
- You use the binary arithmetic operators `+`, `–`, `*`, `/`, and `%` to manipulate values in your programs. Multiplication, division, and remainder always take place prior to addition or subtraction in an expression, unless you use parentheses to override the normal precedence. C# provides several shortcut arithmetic operators, including the binary operators `+=`, `–=`, `*=`, `/=`, and the unary prefix and postfix increment (`++`) and decrement (`--`) operators.
- A `bool` variable can hold only one of two values—true or false. C# supports six comparison operators: `>`, `<`, `>=`, `<=`, `==`, and `!=`. An expression containing a comparison operator has a Boolean value.
- When you perform arithmetic with variables or constants of the same type, the result of the arithmetic retains that type. When you perform arithmetic operations with operands of different types, C# chooses a unifying type for the result and implicitly converts nonconforming operands to the unifying type. You may explicitly override the unifying type imposed by C# by performing a cast.
- You use the `char` data type to hold any single character. You place constant character values within single quotation marks. You can store any character—including nonprinting characters such as a backspace or a tab—in a `char` variable. To store these characters, you must use an escape sequence, which always begins with a backslash.
- In C#, you use the `string` data type to hold a series of characters. The value of a `string` is expressed within double quotation marks. Although the `==` and `!=` comparison operators can be used with `string`s that are assigned literal values, you also can use the `Equals()`, `Compare()`, `CompareTo()`, and `Substring()` methods and the `Length` property that belong to the `String` class.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- Named constants are program identifiers whose values cannot change. Conventionally, their identifiers are all uppercase, with underscores inserted for readability. Enumerations are lists of named constants. The constants are integers by default, and frequently the list holds consecutive values.
- You can use the `ReadLine()` method to accept user input. Often you must use a `Convert` class method to change the input string into a usable data type.

Key Terms

Constant describes data items whose values are fixed.

A **literal constant** is a value that is taken literally at each use.

A **variable** is a named location in computer memory that can hold different values at different points in time.

A **data type** describes the format and size of a data item and defines what types of operations can be performed with the item.

Intrinsic types of data are basic types; C# provides 15 intrinsic types.

An **alias** is another name for something.

A **simple type** is one of the following in C#: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `char`, and `bool`.

Unicode is a 16-bit coding scheme for characters.

A **variable declaration** is the statement that names a variable; it includes the data type that the variable will store, an identifier that is the variable's name, an optional assignment operator and assigned value when you want a variable to contain an initial value, and an ending semicolon.

The **assignment operator** is the equal sign (=); any value to the right of the assignment operator is assigned to, or taken on by, the variable to the left.

An **initialization** is an assignment made when a variable is declared.

An **assignment** is a statement that provides a variable with a value.

To **concatenate** a string with another value is to join the two values end to end using a plus sign.

A **format string** is a string of characters that controls the appearance of output.

Integral data types are those that store whole numbers.

The nine integral types are **`byte`**, **`sbyte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, and **`char`**. The first eight always represent whole numbers, and the ninth type, `char`, is used for characters like 'A' or `'a'`.

Integers are whole numbers.

A **floating-point** number is one that contains decimal positions.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Significant digits provide a measure of the mathematical accuracy of a value.

A **`float`** data type can hold a floating-point number with as many as seven significant digits of accuracy.

A **`double`** data type can hold a floating-point number with 15 or 16 significant digits of accuracy.

The **`decimal`** data type is a floating-point type that has a greater precision and a smaller range than a `float` or `double`, which makes it suitable for financial and monetary calculations.

Scientific notation is a means of expressing very large and small numbers using an exponent.

Standard numeric format strings are strings of characters expressed within double quotation marks that indicate a format for output.

The **format specifier** in a format string can be one of nine built-in format characters that define the most commonly used numeric format types.

The **precision specifier** in a format string controls the number of significant digits or zeros to the right of the decimal point.

A *C#* **culture** is a set of rules that determines how culturally dependent values such as money and dates are formatted.

Arithmetic operators are used to perform arithmetic; they include +, -, *, /, and %.

Operands are the values that operators use in expressions.

Binary operators use two operands—one value to the left of the operator and another value to the right of it.

Operator precedence determines the order in which parts of a mathematical expression are evaluated.

Order of operation is another term for operator precedence.

Associativity specifies the order in which a sequence of operations with the same precedence are evaluated.

The **add and assign operator** (+=) adds the operand on the right to the operand on the left and assigns the result to the operand on the left in one step.

The **prefix increment operator** (++ before a variable) increases the variable's value by 1 and then evaluates it.

The **postfix increment operator** (++ after a variable) evaluates a variable and then adds 1 to it.

Unary operators are operators used with one operand.

The **decrement operator** (`--`) reduces a variable's value by 1. There is a prefix and a postfix version.

A **Boolean variable** can hold only one of two values—true or false.

The **`bool`** data type holds a Boolean value.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

A **comparison operator** compares two items; an expression containing a comparison operator has a Boolean value.

A **unifying type** is the type chosen for an arithmetic result when operands are of dissimilar types.

Implicitly means automatically.

Type precedence is a hierarchy of data types used to determine the unifying type in arithmetic expressions containing dissimilar data types.

An **implicit conversion** or **implicit cast** is the automatic transformation that occurs when a value is assigned to a type with higher precedence.

Explicitly means purposefully.

An **explicit cast** purposefully assigns a value to a different data type; it involves placing the desired result type in parentheses followed by the variable or constant to be cast.

The **char** data type can hold any single character; a literal character is contained between single quotation marks.

An **escape sequence** is two symbols beginning with a backslash that represent a nonprinting character such as a tab.

Hexadecimal, or **base 16**, is a mathematical system that uses 16 symbols to represent numbers.

The **string** data type is used to hold a series of characters; a literal string is contained between double quotation marks.

Lexically means alphabetically.

Immutable describes unchangeable objects such as strings that refer to a new memory location when a new value is assigned.

A **named constant** is an identifier whose value must be assigned upon declaration and whose contents cannot change.

A **self-documenting** program element is one that is self-explanatory.

An **enumeration** is a set of constants represented by identifiers.

Type-safe describes a data type for which only appropriate behaviors are allowed.

An **interactive program** is one that allows user input.

A **prompt** is an instruction to the user to enter data.

To **parse** an item is to break it into component parts.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Review Questions

1. When you use a number such as 45 in a C# program, the number is a ____________________.

   a. literal constant
   b. figurative constant
   c. literal variable
   d. figurative variable

2. A variable declaration must contain all of the following *except* a(n) ____________________.

   a. data type
   b. identifier
   c. assigned value
   d. ending semicolon

3. Which of the following is true of variable declarations?

   a. Two variables of the same type can be declared in the same statement.
   b. Two variables of different types can be declared in the same statement.
   c. Two variables of the same type must be declared in the same statement.
   d. Two variables of the same type cannot coexist in a program.

4. Assume that you have two variables declared as `int var1 = 3;` and `int var2 = 8;`. Which of the following would display 838?

   a. `WriteLine("{0}{1}{2}", var1, var2);`
   b. `WriteLine("{0}{1}{0}", var1, var2);`
   c. `WriteLine("{0}{1}{2}", var2, var1);`
   d. `WriteLine("{0}{1}{0}", var2, var1);`

5. Assume that you have a variable declared as `int var1 = 3;`. Which of the following would display *X 3X?*

   a. `WriteLine("X{0}X", var1);`
   b. `WriteLine("X{0,2}X", varl);`
   c. `WriteLine("X{2,0}X", varl);`
   d. `WriteLine("X{0}{2}", varl);`

6. Assume that you have a variable declared as `int varl = 3;`. What is the value of `22 % var1?`

   a. 0
   b. 1
   c. 7
   d. 21

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

7. Assume that you have a variable declared as `int var1 = 3;`. What is the value of `22 / var1`?

   a. 1 c. 7.333
   b. 7 d. 21

8. What is the value of the expression `4 + 2 * 3`?

   a. 0 c. 18
   b. 10 d. 36

9. Assume that you have a variable declared as `int var1 = 3;`. If `var2 = ++var1`, what is the value of `var2`?

   a. 2 c. 4
   b. 3 d. 5

10. Assume that you have a variable declared as `int var1 = 3;`. If `var2 = var1++`, what is the value of `var2`?

    a. 2 c. 4
    b. 3 d. 5

11. A variable that can hold the two values `true` and `false` is of type ______________________.

    a. `int` c. `char`
    b. `bool` d. `double`

12. Which of the following is *not* a C# comparison operator?

    a. => c. ==
    b. != d. <

13. What is the value of the expression `6 >= 7`?

    a. 0 c. true
    b. 1 d. false

14. Which of the following C# types *cannot* contain floating-point numbers?

    a. `float` c. `decimal`
    b. `double` d. `int`

15. Assume that you have declared a variable as `double hourly = 13.00;`. What will the statement `WriteLine(hourly);` display?

    a. 13 c. 13.00
    b. 13.0 d. 13.000000

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

16. Assume that you have declared a variable as `double salary = 45000.00;`. Which of the following will display *$45,000*?

    a. `WriteLine(salary.ToString("c"));`

    b. `WriteLine(salary.ToString("c0"));`

    c. `WriteLine(salary);`

    d. two of these

17. When you perform arithmetic operations with operands of different types, such as adding an `int` and a `float`, ____________________.

    a. C# chooses a unifying type for the result

    b. you must choose a unifying type for the result

    c. you must provide a cast

    d. you receive an error message

18. Unicode is____________________.

    a. an object-oriented language

    b. a subset of the C# language

    c. a 16-bit coding scheme

    d. another term for hexadecimal

19. Which of the following declares a variable that can hold the word *computer*?

    a. `string device = 'computer';`

    b. `string device = "computer";`

    c. `char device = 'computer';`

    d. `char device = "computer";`

20. Which of the following compares two string variables named `string1` and `string2` to determine if their contents are equal?

    a. `string1 = string2`

    b. `string1 == string2`

    c. `Equals.String(string1, string2)`

    d. `two of the above`

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Exercises

Programming Exercises

1. What is the numeric value of each of the following expressions, as evaluated by the C# programming language?

   a. 2 + 5 * 3
   b. 9 / 4 + 10
   c. 10 / 3
   d. 21 % 10
   e. (5 – 1) * 3
   f. 37 / 5
   g. 64 % 8
   h. 5 + 2 * 4 – 3 * 4
   i. 3 * (2 + 5) / 5
   j. 28 % 5 – 2
   k. 19 / 2 / 2
   l. 28 / (2 + 4)

2. What is the value of each of the following Boolean expressions?

   a. 5 > 4
   b. 3 <= 3
   c. 2 + 4 > 5
   d. 6 == 7
   e. 2 + 4 <= 6
   f. 3 + 4 == 4 + 3
   g. 1 != 2
   h. 2 != 2
   i. –5 == 7 - 2
   j. 3 + 9 <= 0

3. Choose the best data type for each of the following, so that no memory storage is wasted. Give an example of a typical value that would be held by the variable, and explain why you chose the type you did.

   a. the number of siblings you have
   b. your final grade in this class
   c. the population of Earth
   d. the number of passengers on a bus
   e. one player's score in a Scrabble game
   f. the year a historical event occurred
   g. the number of legs on an animal
   h. one team's score in a Major League Baseball game

4. In this chapter, you learned that although a `double` and a `decimal` both hold floating-point numbers, a `double` can hold a larger value. Write a C# program named **DoubleDecimalTest** that declares and displays two variables—a `double` and a `decimal`. Experiment by assigning the same constant value to each variable

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

so that the assignment to the `double` is legal but the assignment to the `decimal` is not. In other words, when you leave the `decimal` assignment statement in the program, an error message should be generated that indicates the value is outside the range of the type `decimal`, but when you comment out the `decimal` assignment and its output statement, the program should compile correctly.

5. Write a C# program named **MilesToKilometers** that declares a named constant that holds the number of kilometers in a mile: 1.6. Also declare a variable to represent a distance in miles, and assign a value. Display the distance in both miles and kilometers—for example, *3 miles is 4.8 kilometers.*

6. Convert the `MilesToKilometers` class to an interactive application named **MilesToKilometersInteractive**. Instead of assigning a value to the `miles` variable, accept the value from the user as input.

7. Write a C# program named **ProjectedRaises** that includes a named constant representing next year's anticipated 4 percent raise for each employee in a company. Also declare variables to represent the current salaries for three employees. Assign values to the variables, and display, with explanatory text, next year's salary for each employee.

8. Convert the `ProjectedRaises` class to an interactive application named **ProjectedRaisesInteractive**. Instead of assigning values to the salaries, accept them from the user as input.

9. Cramer Car Rental charges $20 per day plus 25 cents per mile. Write a program named **CarRental** that prompts a user for and accepts a number of days and miles driven and displays the total rental fee.

10. Write a program named **HoursAndMinutes** that declares a `minutes` variable to represent minutes worked on a job, and assign a value to it. Display the value in hours and minutes. For example, 197 minutes becomes 3 hours and 17 minutes.

11. Write a program named **Eggs** that declares four variables to hold the number of eggs produced in a month by each of four chickens, and assign a value to each variable. Sum the eggs, then display the total in dozens and eggs. For example, a total of 127 eggs is 10 dozen and 7 eggs.

12. Modify the `Eggs` program to create a new one named **EggsInteractive** that prompts the user for and accepts a number of eggs for each chicken.

13. Write a program named **Dollars** that calculates and displays the conversion of an entered number of dollars into currency denominations—*20s*, *10s*, *5s*, and *1s*.

14. Write a program named **Tests** that declares five variables to hold scores for five tests you have taken, and assign a value to each variable. Display the average of the test scores to two decimal places.

15. Modify the **Tests** program to create a new one named **TestsInteractive** that accepts five test scores from a user.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

16. Write a program named **FahrenheitToCelsius** that accepts a temperature in Fahrenheit from a user and converts it to Celsius by subtracting 32 from the Fahrenheit value and multiplying the result by 5/9. Display both values.

17. Write a program named **Payroll** that prompts the user for a name, Social Security number, hourly pay rate, and number of hours worked. In an attractive format (similar to Figure 2-23), display all the input data as well as the following:
    - gross pay, defined as hourly pay rate times hours worked
    - federal withholding tax, defined as 15 percent of the gross pay
    - state withholding tax, defined as 5 percent of the gross pay
    - net pay, defined as gross pay minus taxes

```
C:\C#\Chapter.02>Payroll
Enter your name: Ruth Roberts
Social Security number: 987-65-4320
Hourly pay rate: 13.50
Hours worked: 36

Payroll Summary for: Ruth Roberts
SSN: 987-65-4320
You worked 36 hours at $13.50 per hour

Gross pay:                $486.00
Federal withholding:       $72.90
State withholding :        $24.30
----------------------------------
Net pay:                  $388.80

C:\C#\Chapter.02>
```

Figure 2-23 Typical execution of the `Payroll` program

18. Create an enumeration named `Month` that holds values for the months of the year, starting with `JANUARY` equal to 1. Write a program named **MonthNames** that prompts the user for a month integer. Convert the user's entry to a `Month` value, and display it.

19. Create an enumeration named `Planet` that holds the names for the eight planets in our solar system, starting with `MERCURY` and ending with `NEPTUNE`. Write a program named **Planets** that prompts the user for a numeric position, and display the name of the planet that is in the requested position.

20. Pig Latin is a nonsense language. To create a word in pig Latin, you remove the first letter and then add the first letter and "ay" at the end of the word. For example, "dog" becomes "ogday" and "cat" becomes "atcay." Write a program named **PigLatin** that allows the user to enter a word and displays the pig Latin version.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Debugging Exercises

1. Each of the following files in the Chapter.02 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, DebugTwo1.cs will become FixedDebugTwo1.cs.
   a. DebugTwo1.cs
   b. DebugTwo2.cs
   c. DebugTwo3.cs
   d. DebugTwo4.cs

Case Problems

1. In Chapter 1, you created two programs to display the motto for the Greenville Idol competition that is held each summer during the Greenville County Fair. Now write a program named **GreenvilleRevenue** that prompts a user for the number of contestants entered in last year's competition and in this year's competition. Display all the input data. Compute and display the revenue expected for this year's competition if each contestant pays a $25 entrance fee. Also display a statement that indicates whether this year's competition has more contestants than last year's.
2. In Chapter 1, you created two programs to display the motto for Marshall's Murals. Now write a program named **MarshallsRevenue** that prompts a user for the number of interior and exterior murals scheduled to be painted during the next month. Compute the expected revenue for each type of mural. Interior murals cost $500 each, and exterior murals cost $750 each. Also display the total expected revenue and a statement that indicates whether more interior murals are scheduled than exterior ones.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Copyright 2016 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.