2 C# Programming Basics

In this chapter we will learn about the basic elements of a simple C# program. We use integer values and variables to represent storage locations. Identifiers name variables and other program elements. The assignment statement lets us assign a value to a variable. Expressions compute values. We study arithmetic expressions built using arithmetic operators. Precedence rules permit less use of parentheses in expressions. Input and output methods communicate between the user and the computer. A method implements a computation or an action. We learn to create our own methods, passing parameters and returning values.

OBJECTIVES

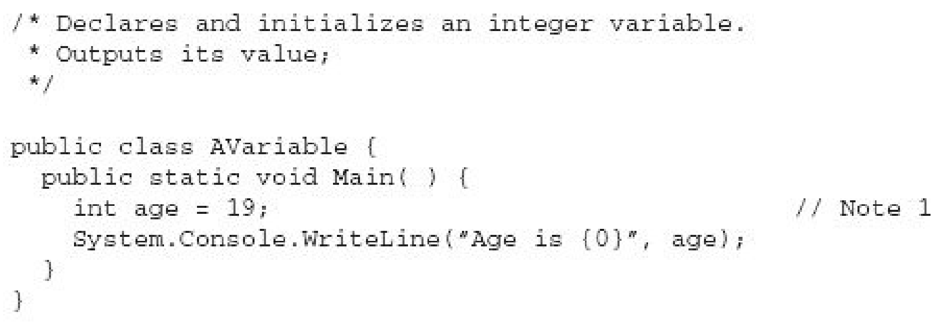
- Learn C# syntax
- Use integer variables and constants
- Use the assignment statement
- Use arithmetic expressions
- Understand operator precedence
- Use methods and parameters

2.1 VARIABLES

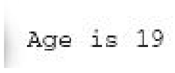
A variable represents a storage location. Every variable has a name and a type that determines what kind of data it holds. In this section, we look at the rules for creating names and introduce the int type, which represents a range of integer values

We use a simple output statement to display the value of a variable. Example 2.1 represents a very simple C# program containing a variable

EXAMPLE 2.1 ■ AVariable.cs



```
/* Declares and initializes an integer variable.
 * Outputs its value;
 */

public class AVariable {
  public static void Main( ) {
    int age = 19;                                // Note 1
    System.Console.WriteLine("Age is {0}", age);
  }
}
```



```
Age is 19
```

Age is 19

Note 1: int age = 19;

age is a local variable, defined inside the Main method for use there. Later we shall see other types of variables

Example 2.1 has a structure similar to that of Example 1.1. It declares the AVariable class. This class declaration contains a declaration for the Main method. The Main method contains two statements. This section discusses the first statement, int age = 19;
which is a local variable declaration, stating that age is a variable of type int with an initial value of 19

The variable age represents a memory location that holds the value 19 initially. Figure 2.1 shows the effect of this declaration. The box signifies a location in the computer's memory. Putting 19 inside the box indicates that this location holds the value 19.[1]

Each variable has a name and a type. We discuss each of these features

Identifiers

An **identifier** names program elements. The identifier age names an integer variable. The identifier AVariable names the class containing our program. We chose

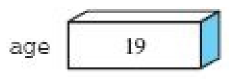


FIGURE 2.1 A variable representing a memory location

these names. Notice that these identifiers include both uppercase and lowercase characters. C# is case sensitive, so identifiers age, Age, and AGE are all different although they each use the same three letters of the alphabet in the same order

Style

Start variable names with a lowercase character. Start method and class names with an uppercase character

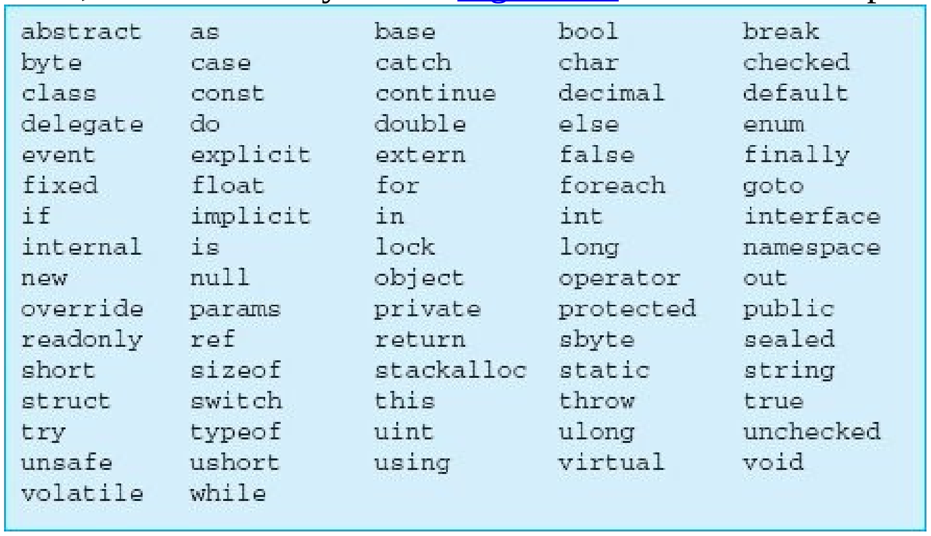
Digits occur in identifiers but cannot be the first character. Identifiers may also use the underscore character and even start with it. For example, *hat*, *hat*box, and My________my are all valid identifiers

Style

Use underscores or uppercase letters to make identifiers easier to read. For example, use a_big_car or aBigCar rather than abigcar. Use meaningful names such as age rather than arbitrary names such as xyz

Keywords

Keywords are identifiers that are reserved for special uses. In Example 2.1, public, class, static, void, and int are keywords. Figure 2.2 shows the complete list of C# keywords



abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

FIGURE 2.2 Keywords

Some valid C# identifiers are

```
savings
textLabel
rest_stop_12
B3
_test
My____my
```

Some invalid identifiers are

```
4you       // Starts with a number
x<y        // Includes an illegal character, <
top-gun    // Includes an illegal character, -
int        // Reserved keyword
```

The Character Set

The character set defines the characters we can use in a program. The ASCII (pronounced as'-key) character set contains 128 printing and nonprinting characters shown in Appendix D. The ASCII characters include uppercase and lowercase letters, digits, and punctuation. For worldwide use, a programming language must have a much bigger character set to include the many characters of the various major languages. C# uses the Unicode character set which contains thousands of characters, including all the ASCII characters. For example, Unicode includes the Greek letter gamma, **Γ**. We will use only the ASCII characters in this book

Type int

We declare every variable, indicating its type. A **type** defines the values that a variable can have. In Example 2.1, the variable age has type int. In C#, type int represents a range of integer values. C# implements it using 32 binary digits (called bits). Example 2.2 displays the largest and smallest values of the int type

EXAMPLE 2.2 ■ MaxMinInt.cs

```
/* Outputs the largest and smallest int values.
 */

public class MaxMinInt {

    // Execution starts here
  public static void Main() {
    int max = int.MaxValue;                         // Note 1
    int min = int.MinValue;
    System.Console.WriteLine
          ("The largest int value is {0}", max);
```

```
      System.Console.WriteLine
             ("The smallest int value is {0}", min);
    }
  }
```

```
The largest int value is 2147483647
The smallest int value is -2147483648
```

Note 1: int max = int.MaxValue;

The keyword int is another name for the type System.Int32 in the .NET framework. Type Int32 declares constants MaxValue and MinValue. We could have referred to this value as System.Int32.MaxValue

Example 2.2 shows that the int type represents values from -2,147,483,648 to 2,147,483,647. C# has many other predefined types, both numerical and nonnumerical, which we will introduce in later chapters. See Appendix E for a list

Initialization

Notice that we specified initial values for each of the variables in Examples 2.1 and 2.2. For example, the statement

int age = 19;

declares that the variable age has type int and has an initial value of 19. C# does not initialize variables declared inside methods (called local variables). Making the declaration

int age;

without initializing age would leave the contents of the variable unknown, as Figure 2.3 shows

The C# compiler will not allow us to use a local variable that has an unknown value. It is a good practice always to initialize local variables

Some characteristics of variables in a programming language are

- Variables can vary in value as the program runs
- Each variable usually has a single purpose in a program

FIGURE 2.3 Memory location for the age variable

So far, we have not allowed our variable to vary. Using the assignment operator in the next section, we will give our variables new values

The BIG Picture

A variable represents a storage location. It has a name and type. We use an identifier to name variables and other program elements. Keywords are reserved and may not be used as identifiers. The int type represents a range of integer values, both positive and negative. We refer to variables we declare inside a method as local variables because they are for use only within that method

Good practice includes an initialization in a variable declaration statement. Write and WriteLine statements allow us to display results

✓ Test Your Understanding

1. Which of the following are valid identifiers? For each nonvalid example, explain whyit is not valid.

a. Baby

b. *chip*eater

c. any.time

d. #noteThis

e. &car

f. GROUP

g. A103

h. 76trombones

i. float

j. intNumber

k. $$help

2. A variable represents a____________________.

3. A variable has a_______________and a______________.

4. The largest value the ___type can hold is 2,147,483,647.

5. How can we improve the following declaration?

int number;

2.2 ASSIGNMENT

We initialize a variable only once, in its declaration. To change the value of a variable after we have declared it, we can use the assignment operator. This, as its name suggests, assigns a value to a variable. C# uses the equal sign, =, for the assignment operator. An example of a simple assignment statement is age = 10; in which we assign the value 10 to the variable age. This assignment statement assumes that we have already declared the variable age. The compiler would report an error if we had not declared age

We declare and initialize a variable only once, but we can assign it a value many times in a program. Later in the program we may wish to change the value of age, say to 20, using the assignment statement

age = 20;

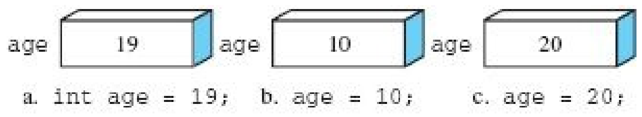


FIGURE 2.4 Declaring and assigning values to a variable

Remember that the variable, age, has one location. Each assignment replaces the old value with the newly assigned value. Figure 2.4 shows the changes resulting from the assignments to age

So far we have only assigned constant values to variables, but we can also assign the value of one variable to another, as in

```
int mySize = 9;
int yourSize = 10;
mySize = yourSize;
```

The assignment takes the value of yourSize, which we initialized to 10, and assigns it to mySize. Figure 2.5 shows the locations for yourSize and mySize before and after the assignment

We can write an arithmetic expression such as y + 2 on the right-hand side of an assignment. The computer will then evaluate the expression and assign that value to the variable on the left-hand side of the assignment. We will learn about arithmetic expressions in Section 2.4. The assignment mySize = yourSize + 2; would give mySize a value of 12, given that yourSize still has the value of 10, with which it was initialized

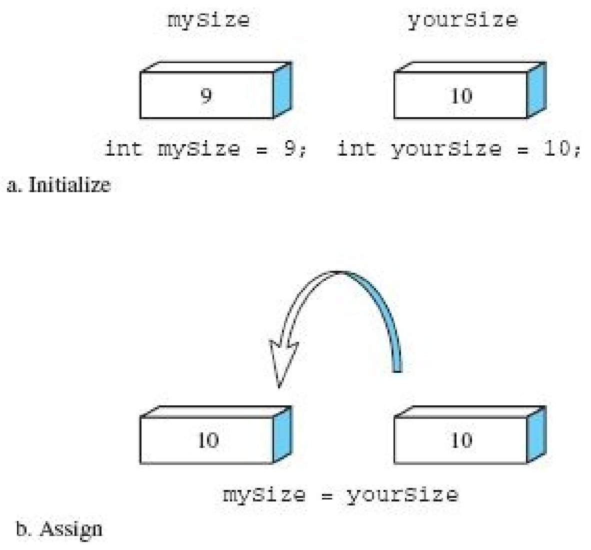


FIGURE 2.5 The result of an assignment

Notice that when a variable occurs on the right-hand side of an assignment we use its value that we find in its storage location. When a variable occurs on the left-hand side of an assignment, we store a value in its memory location. Variables perform this very important function of storing values for later use in our program

Constructing a Simple Program

Now we have the tools to write a simple C# program. We can store values in variables using initialization and assignment; we can use the values of variables and constants on the right-hand side of assignments and in Write and WriteLine statements. In a program, we might group several of these statements together for C# to execute one after the other, as in

```
int number1 = 25;
Console.WriteLine(number1);
```

which declares an integer variable, number1, initializes it to 25, and displays its value on the screen

A C# program consists, roughly speaking, of one or more class definitions. To write a C# program, we pick a name for a class. Example 2.3 uses the name AssignIt, which starts, in good C# style, with a capital letter. The program structure will then look like

```
public class AssignIt {

    //  The program code goes here

}
```

All the code will go inside the curly braces after the class name. For now we include a comment using the double forward slash, //. C# ignores anything on the line after the //; it is just a comment for human readers. We declare our class public so everyone can use it. We must always use the public modifier when declaring the Main method. We will discuss different types of access later in the text C# organizes executable code by placing it into methods.[2] A method contains code to perform an operation. We often name methods to represent their function. For example, in Section 2.4 we define a MultiplyBy4 method to multiply a value by 4. As we have seen, C# uses a special method named Main to start processing a program. The Main method might be better named as StartUp, but it retains the name Main used in the C and C++ programming languages Because C# starts up an application using a Main method, we add a Main method to our class, so it now looks like this:

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



     /*  code goes
         here
      */
  }
}
```

C# also ignores anything between /* and the next */, so we can use these symbols to enclose comments that take more than one line. We place the code for the Main method between braces, {}, which contain the code comprising the body of the method The Main method is itself inside a class named AssignIt. Figure 2.6 shows a class diagram for the AssignIt class. We use the Unified Modeling Language (UML).[3] The class diagram has three parts. The top section contains the class name. The middle section contains state variables, which we discuss in Chapter 5. The bottom section contains method names We use the modifier static to indicate that the Main method stays with the class. Following UML notation, we underline (or italicize) static methods to distinguish them from methods that will be part of object instances

Now that we have the framework, we write some code that we insert in the Main method. Let us declare and use some integer variables, trying out initialization, assignment, and displaying our results. Example 2.3 shows this program Every C# program must have the .cs extension, so we use the filename AssignIt.cs for Example 2.3

Style

Formatting programs nicely makes them easier to understand and modify. Example 2.3 contains a comment prefacing the AssignIt class and a comment before the Main method. We should comment each class and each method. We should comment any section of code that needs explanation of its purpose or the way it works. Because the text examples are for teaching purposes, the comments include more details than might otherwise occur. To avoid including cumbersome comments directly in the code, the text examples include a simple Note comment with the text of the comment appearing after the program Example 2.3 indents the Main method with respect to the class containing it. It indents the code within the Main method. It has no long lines. We break long lines as needed to preserve the margins and the indentation

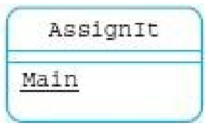


FIGURE 2.6 The AssignIt class

EXAMPLE 2.3 ■ Assignlt.cs

```
/*   Declares two integer variables with initial values.
 *   Assigns a new value to the second variable.
 *   Outputs the new value.
 */

public class AssignIt {

      // Illustrates assignment
  public static void Main() {                        // Note 1
    int  number1 = 25;                               // Note 2
    int  number2 = 12;
    number2 = number1 +  15;                         // Note 3
    System.Console.WriteLine("Number2 is now {0}", number2);
  }
}
```

Output

```
Number2 is now 40
```

Note 1: public static void Main() {

C# starts execution of a program with the code in the Main method.

We must have a method named Main. We discuss the choice of parameters and return type for the Main method later in the text

Note 2: int number1 = 25;

We use the keyword int to declare an integer variable. Here we declare number1 to be an integer variable and give it an initial value of 25

Note 3: number2 = number1 + 15;

This is an assignment statement. We compute the value of the expression, number1 + 15, on the right side of the equals sign and assign it to the variable, number2, on the left side. This value, 40, will replace whatever previous value number2 had, which was 12 in this example.

Constants

We changed the value of variable number2 in Example 2.3. We can define constants, which we cannot change. For example, the declaration

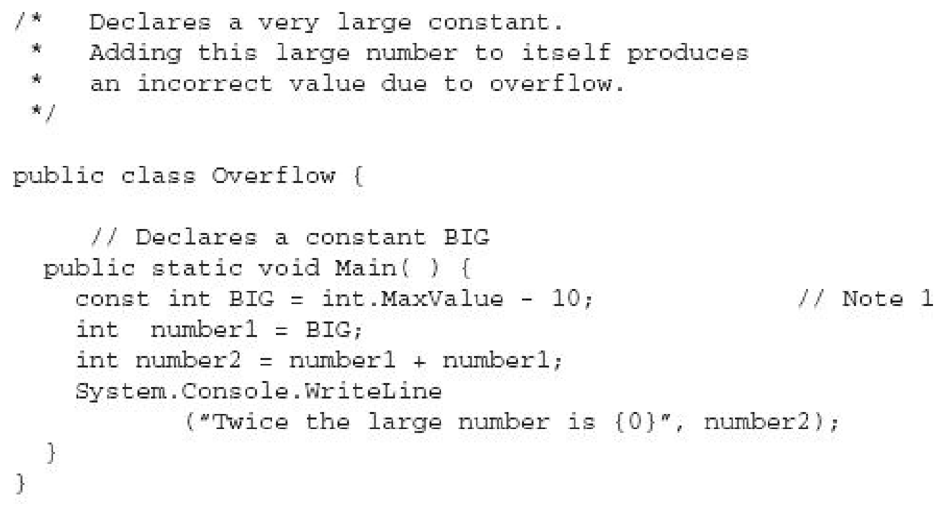
```
const int FIXED = 2 5;
```

declares a constant named FIXED. Trying to assign a value to FIXED would generate a compilation error. The modifier const means that the value of FIXED is final and cannot be changed. We use a constant to highlight an important difference between computing and mathematics in Example 2.4

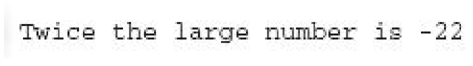
Style

Using uppercase letters for constants makes them easy to distinguish from variables in our code

EXAMPLE 2.4 ■ Overflow.cs



```
/*   Declares a very large constant.
 *   Adding this large number to itself produces
 *   an incorrect value due to overflow.
 */

public class Overflow {

     // Declares a constant BIG
  public static void Main( ) {
    const int BIG = int.MaxValue - 10;              // Note 1
    int  number1 = BIG;
    int number2 = number1 + number1;
    System.Console.WriteLine
           ("Twice the large number is {0}", number2);
  }
}
```



```
Twice the large number is -22
```

Twice the large number is -22

Note 1: const int BIG = int.MaxValue - 10;

This value is 2147483637. The compiler will generate an error if we try to assign a value to BIG later in the program

Mathematically, the output should be 4,294,967,274, but the int type cannot represent a number this large, just as in Figure 2.7, two chairs cannot seat three people. We do not need to understand the details of what happens when we compute a number too large to represent. We call this phenomenon **overflow** because the number overflows the space available for it. An int variable holds 32 bits and

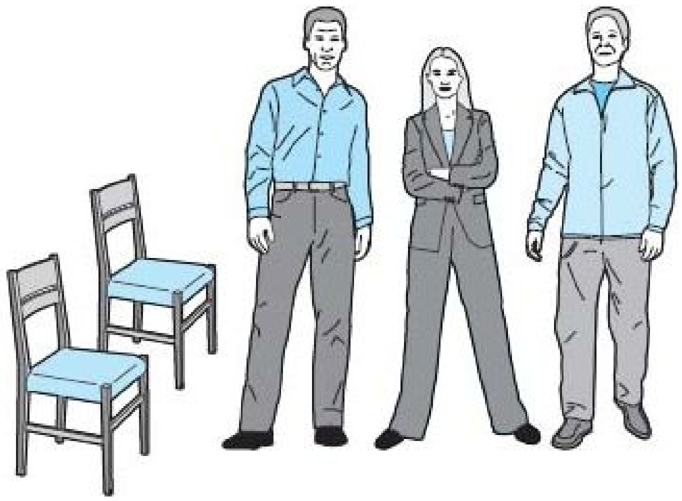


FIGURE 2.7 Overflow—Who does not get a seat?

some integers require more than 32 bits. The moral is that not everything the computer produces is necessarily correct. One must inspect results carefully

The BIG Picture

An assignment statement assigns a value to a variable. We retrieve the value of a variable used on the right side of an assignment statement. We store a value in the location represented by a variable on the left side of an assignment

A C# program consists of one or more class definitions. C# starts execution with an application's Main method. In our examples so far, the Main method contains a sequence of statements. These are declaration, assignment, or output statements.

We cannot change a constant

✓ Test Your Understanding

Try It Yourself ➤

6. Compile and run Example 2.3 to check that it works properly

7. In Example 2.3, what is the largest value we can use for the variable number1 ? What is the smallest?

Try It Yourself ➤

8. Try omitting the declaration of the variable number1 in Example 2.3 and see what error you get

2.3 INPUT AND OUTPUT

The programs in the last section specify the values for variables and constants in the program. To change the initialization

int number1 = 25;

to use the number 30 instead, we would have to edit the program, making the change from 25 to 30, and recompile it again. In this section, we learn to enter values from the keyboard so we can run the program again with different input values without the need to recompile. We used the WriteLine statement without much explanation in the last section, but here we will explore possibilities for output

Inputting from the Console

The term **console** reminds older veterans of the early computers that could only display characters and not graphics. We use it now to refer to a window that only displays characters and not graphics

We wish to enter data from the keyboard into the console. A good program will prompt the user with a message describing the data to enter. For example, the statement

```
System.Console.Write("Enter your name: ");
```

displays the message

```
String name = System.Console.ReadLine();
```

and remains on the same line, where the user will enter his or her name.

The statement

```
String name = System.Console.ReadLine();
```

returns a line of text that the user enters by typing the characters on the keyboard and pressing the *Enter* key. When the executing program reaches this ReadLine method, it waits for the user to enter a line of text, which it returns to the program, assigning it to the variable name. The String type represents text. We will learn more about it later. Just to verify that we have received the data, we use the WriteLine method to display it in the console.

EXAMPLE 2.5 InputName.cs

```
// Inputs text from the keyboard.
using System;                                      // Note 1
public class InputName {
  public static void Main( ) {
    Console.Write("Enter your name: ");            // Note 2
    String name = Console.ReadLine();              // Note 3
    Console.WriteLine("    Your name is {0}", name);
  }
}
```

```
Enter your name: Art
    Your name is Art
```

Note 1: using System;

The System namespace includes the Console class containing the Write, ReadLine, and WriteLine methods which we use in this example. This

line is a using directive. It lets us use the classes in the System namespace without the System prefix. We can use the Console class without prefixing it with System . Without this directive, we must use System.Console to refer to the Console class in the System namespace. Using the directive, we have less keyboarding to do and shorter lines

Note 2: Console.Write("Enter your name: ");

The Write statement leaves the cursor on the current line. By contrast, the WriteLine statement moves the cursor to the beginning of the next line. The cursor signals the position for the next input or output operation

Note 3: String name = Console.ReadLine();

When execution reaches the ReadLine statement it waits for the user to enter text and press the *Enter* key

Inputting an Integer

The ReadLine method inputs text. If we enter 23, it will return the string "23". We are used to reading base 10 numerals, and readily understand this as a symbol for three more than the normal number of fingers and toes combined. Ancient Romans would have written XXIII to represent the same number. AC#

program does not use either of these representations for integers. The int type has a Parse method that converts a base 10 representation to the 32-bit binary representation it uses

EXAMPLE 2.6 InputInteger.cs

```
/*   Inputs a base 10 numeral from the console.
 *   Converts to an int and outputs twice its value.
 */

using System;
public class InputInteger {
  public static void Main( ) {
    Console.Write("Enter an integer: ");
    int  number1 = int.Parse(Console.ReadLine());
                                                       // Note 1
    int number2 = number1 + number1;
    Console.WriteLine
          ("    Twice the number  is {0}", number2);
  }
}
```

```
 Enter an integer: 23
     Twice the number is 46
```

Note 1: int number1 = int.Parse(Console.ReadLine());

Type int is a shorthand for System.int32, so we could have written this method as Int32.Parse.

When we cover user interfaces later, we will have many other ways for the user to input and interact

Output to the Console

We have used the WriteLine and Write methods to write to the console. The invocation

```
System.Console.WriteLine("The number is: {0}", number1);
```

has two arguments. The first is a string and the second is a value that C# will convert to a string for display. The format specifier, {0}, indicates the position where the second argument will appear in the string

To output more than one number we use additional format specifiers, {0}, {1}, {2}, and so on. For each specifier, we include an argument to replace it in the output string. Thus if number1 is 35 and number2 is 70, the statement

```
Console.WriteLine("The number is {0} and twice it is {1}.",
                   number1, number2);
```

will output

```
The number is 35 and twice it is 70.
```

We can output the data in any order. For example,

```
Console.WriteLine
        ("Twice the number is {1} and the number is {0}",
          number1, number2);
```

outputs

```
Twice the number is 70 and the number is 35.
```

The numbers 0 and 1 in {0} and {1} indicate an argument position after the format string in the WriteLine invocation. We can also specify a specific format. For example, {0:C} would display as currency. In the United States, 35 dollars displays as $35.00

EXAMPLE 2.7 OutputFormat.cs

```
// Outputs to the console.
using System;
public class OutputFormat {
  public static void Main( ) {
    int number1 = 35;


      int number2 = number1 + number1;
      Console.WriteLine
               ("The number is {0} and twice it is {1}.",
                 number1, number2);
      Console.WriteLine
               ("{0} is the number and {1} is its double.",
                 number1, number2);
      Console.WriteLine
              ("Twice the number is {1} and the number is {0}",
                 number1, number2);
      Console.WriteLine
               ("The first is {0:C} and twice it is {1:C}.",
                 number1, number2);
    }
  }
```

```
The number is 35 and twice it is 70.
35 is the number and 70 is its double.
Twice the number is 70 and the number is 35.
The first is $35.00 and twice it is $70.00.
```

Outputting a Table

We would like to output a table in which the columns have a fixed width. We might want to align names to the left of the column and numbers to the right. To achieve this format we use the specification {0, -10} to left-align names in a field of length 10, and {1, 10} to right-align numbers in a field of size 10

EXAMPLE 2.8 OutputTable.cs

```
/*  Formats a table with fixed column lengths and
 *  either right or left alignment.
 */

using System;
public class OutputTable {
  public static void Main( ) {
    Console.WriteLine("{0,-10}{1,10}", "Names", "Numbers" );
    Console.WriteLine
            ("{0,-10}{1,10}", "Sheila", 12345);
                                                // Note 1
    Console.WriteLine("{0,-10}{1,10}", "Frances", 241);
    Console.WriteLine("{0,-10}{1,10}", "Michael", 4141);
  }
}
```

```
Names        Numbers
Sheila         12345
Frances          241
Michael         4141
```

Note 1: `Console.WriteLine`

```
    ("{0,-10}{1,10}", "Sheila", 12345);
```

We can output literals as well as values of variables. A **literal** is a source code representation of a value. Here "Sheila" is a string and 12345 is an integer

The BIG Picture

Inputting from the console allows us to rerun the same program with different data. We can format output to make it more useful and visually appealing

Test Your Understanding

9. Enter *hat* instead of an integer when Example 2.6 requests a number. What happens?

10. Rewrite Example 2.5 without the using directive. Compile and run the modified version

11. If *x* is 53 and *y* is 8786, what will the method display?

System.Console.WriteLine(“You owe {1:C} which is {0} days overdue”, x, y);

2.4 ARITHMETIC EXPRESSIONS

In the process of solving problems, we perform operations on the data. Each type of data has suitable operations associated with it. Integer data in C#, which we use in this chapter, has the familiar arithmetic operations addition, subtraction, multiplication, division, negation, and a remainder operation A **binary** operator such as ‘+' takes two operands, as in the expression 3 + 4, where the numbers 3 and 4 are the operands. C# supports these binary arithmetic operators:

```
+    addition
-    subtraction
*    multiplication
/    division
%    remainder
```

A **unary** operator takes one operand as in the expression -3. C# supports these unary arithmetic operators:

```
-   negation
+   (no effect)
```

If the operands are integers, then the result of an arithmetic operation will be an integer. Addition, subtraction, and multiplication behave as we expect from ordinary arithmetic. Some examples are:

Operation	**Result**
32 + 273	305
63 - 19	44

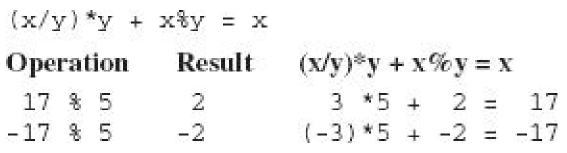
42 * 12 504

Integer division produces an integer result, truncating toward zero if necessary, meaning that it discards the fractional part, if any

Operationt	**Result**	
12 / 3	4	Exact
17 / 5	3	Discards the 2/5
-17 / 5	-3	Discards the -2/5

The operation x%y computes the remainder when x is divided by y (see Figure 2.8). The remainder operation obeys the rule

(x/y)*y + x%y = x



```
(x/y)*y + x%y = x
```

Operation	**Result**	**(x/y)*y + x%y = x**
17 % 5	2	3 *5 + 2 = 17
-17 % 5	-2	(-3)*5 + -2 = -17

Examples of the unary operations are -7, which negates the seven, and +3. In summary, Figure 2.9 shows the C# arithmetic operations for integers

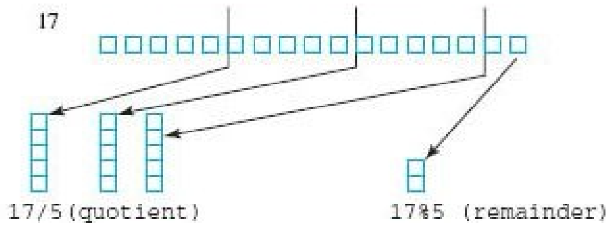
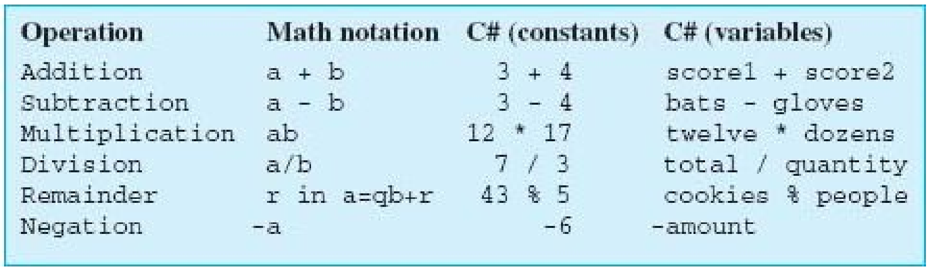


FIGURE 2.8 Integer division and remainder



Operation	Math notation	C# (constants)	C# (variables)
Addition	a + b	3 + 4	score1 + score2
Subtraction	a - b	3 - 4	bats - gloves
Multiplication	ab	12 * 17	twelve * dozens
Division	a/b	7 / 3	total / quantity
Remainder	r in a=qb+r	43 % 5	cookies % people
Negation	-a	-6	-amount

FIGURE 2.9 C# arithmetic operations

EXAMPLE 2.9 Arithmetic.cs

```
/* Try out arithmetic operators on integer data.
 */

public class Arithmetic {

    // Illustrates +, -, *, /, and %
  public static void Main () {
    int x = 25;
    int y = 14;
    int z = x + y;                                   // Note 1
    int w = x - y;
    int p = -y;
    System.Console.WriteLine
     ("x + y = {0}       x - y = {1}         -y = {2}",
       z, w, p);
    z = x * y;                                       // Note 2
    w = x / 7;                                       // Note 3
    p = x % 7;
    System.Console.WriteLine
     ("x * y = {0}      x / 7 = {1}       x % 7 = {2}",
       z, w, p);
  }
}
```

```
x + y = 39     x - y = 11    -y = -14
x * y = 350    x / 7 = 3      x % 7 = 4
```

Note 1: int z = x + y;

We initialize z with the value of the expression x + y. Because we have already initialized x and y, the value of x + y is well defined

Note 2: z = x * y;;

In the next three lines we reuse the variables z, w, and p, giving them new values and illustrating the use of the operators *, /, and %

Note 3: w = x / 7;

Recall that integer division produces an integer result, so 25 / 7 = 3

Test Your Understanding

12. If a=4, b=23, c = -5, and d=61, evaluate

a. b/a

b. b%a

c. a%b
d. b/c
e. c*d
f. d%b
g. c/a
h. c%a

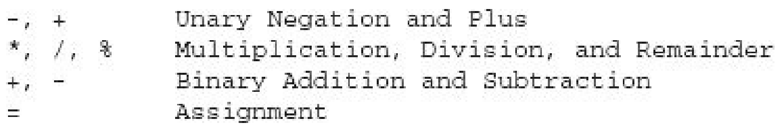
Try It Yourself

13. Change the variable initializations in Example 2.9 to x=12 and y=5. What output do you expect from this modified program? Compile and run it to see if your ex pectations are correct

Precedence of Arithmetic Operators

In mathematics we apply some common rules to decide how each operation gets its operands. For example, in the expression 3 + 4*5 we would multiply 4*5, giving 20, and then add 3+20 to get the result of 23. We say the multiplication has higher precedence than addition, meaning that it gets its operands first. In Figure 2.10 we show that *gets its operands first by drawing a box around the expression 4*5

If we want to do the addition first, we would need to use parentheses, as in (3 + 4)*5, as shown in Figure 2.11. We compute everything inside parentheses first, so we would add 3 + 4, giving 7, and then multiply 7*5, giving 35. By remembering the rules of precedence, we can often avoid using parentheses. We could have written parentheses in the original expression, which would then be 3+(4*5), but these parentheses are not needed because we know that multiplication goes first **Higher Precedence**



-, +	Unary Negation and Plus
*, /, %	Multiplication, Division, and Remainder
+, -	Binary Addition and Subtraction
=	Assignment

We evaluate x-3/y as x-(3/y) because '/' has higher precedence than '-'. We evaluate -7 + 10 as (-7)+10 or 3, because negation has higher precedence than

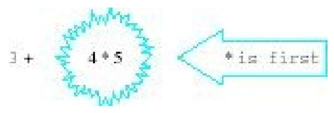


FIGURE 2.10 Multiplication gets its operands first

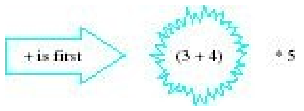
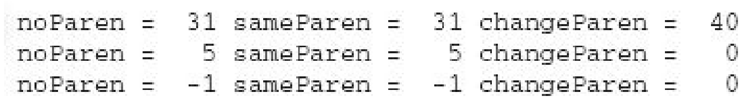


FIGURE 2.11 Compute within parentheses first

addition. In the case of arithmetic operators of equal precedence, we evaluate from left to right. Thus, we compute 3+x+7 as (3+x)+7 and 10-7-5 as (10-7)-5, which is -2. We say that the *associativity* of the binary arithmetic operators is from left to right.

EXAMPLE 2.10 Precedence.cs

```
/* Illustrates precedence rules for arithmetic operators.
 */

public class Precedence {

      /* Computes expressions three ways
       *    Without parentheses --- uses precedence rules
       *    With parentheses the same as precedence rules
       *    With parentheses different than precedence rules
       */
  public static void Main() {
    int a = 3;                                     // Note 1
    int b = 4;
    int c = 5;
    int noParen = a + 7 * b;
    int sameParen = a + (7 * b);
    int changeParen = (a + 7) * b;
    System.Console.WriteLine
      ("noParen = {0,3} sameParen = {1,3} changeParen = "
           + "{2,3}", noParen, sameParen, changeParen);
    noParen = c / a + 4;                           // Note 2
    sameParen = (c / a) + 4;
    changeParen = c / (a + 4);
    System.Console.WriteLine
      ("noParen = {0,3} sameParen = {1,3} changeParen = "
           + "{2,3}", noParen, sameParen, changeParen);
    noParen = c - a % b - a;                       // Note 3
    sameParen = (c - (a % b)) - a;
    changeParen = (c - a) % (b - a);
    System.Console.WriteLine
      ("noParen = {0,3} sameParen = {1,3} changeParen = "
           + "{2,3}", noParen, sameParen, changeParen);
  }
}
```



```
noParen =  31 sameParen =  31 changeParen =  40
noParen =   5 sameParen =   5 changeParen =   0
noParen =  -1 sameParen =  -1 changeParen =   0
```

Note 1: int a = 3;

We use the variable noParen to compute the value of an expression without parentheses. The precedence order will determine which operation to perform first. The variable sameParen shows the expression fully parenthesized, but computed in the same order specified by the precedence. Thus the values of noParen and sameParen should be equal. The variable changeParen computes the same expression, but now with parentheses placed to change the order of evaluation to be different than the order specified by precedence. Thus, the value of changeParen may be different from noParen

Note 2: noParen = c / a + 4;

Recall that integer division gives an integer value so 5/3 = 1

Note 3: noParen = c - a % b - a;

Recall that '%' is the remainder operator, so 3%4 = 3

Try It Yourself ➤

✓ Test Your Understanding

14. Change the variable initializations in Example 2.10 to a=7, b=3, and c = -2. What output do you expect from this modified program? Compile and run it to see if your expectations are correct.

15. Evaluate the following C# expressions, where x=2, y=3, z = -4, and w=5.

a. x + w / 2

b. z * 4 - y

c. y + w % 2

d. x + y - z

e. x * z / y

f. x + z * y / w

g. y * x - z / x

h. w * x % y - 4

i. 14 % w % y

16. Insert parentheses in each expression in problem 15, following the C# operator precedence order. This will show what you would have to write if C# did not use precedence rules. For example, inserting parentheses in x+w*z+3 gives (x+(w*z))+3, because ‘*' has the highest precedence, and C# evaluates the left ‘+' first.

Combining Assignment and Arithmetic[4]

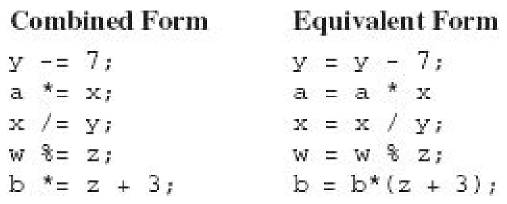
Suppose we want to add 5 to a variable x. We could do that with the assignment statement

x = x + 5;

C# has an operator that combines the assignment and the addition into one operator, +=. We can write the preceding statement more simply as

X += 5;

C# also has operators that combine assignment with the other arithmetic operators: -=, *=, /=, and %=. We must enter these two-character operators without any space between the two symbols. Some examples are in the following list



Combined Form	Equivalent Form
`y -= 7;`	`y = y - 7;`
`a *= x;`	`a = a * x`
`x /= y;`	`x = x / y;`
`w %= z;`	`w = w % z;`
`b *= z + 3;`	`b = b*(z + 3);`

Note in the last example that we put parentheses around the entire right-hand side expression, z + 3, and multiplied that entire expression by the left-hand side variable, b

EXAMPLE 2.11 AssignOps.cs

```
/*  Uses the operators that combine arithmetic and
 *  assignment
 */

using System;
public class AssignOps {

     // Illustrates -=, *=, /=, and %=
  public static void Main() {
    int a = 2;                                    // Note 1
    int b = 4;
    int x = 3;
    int y = 5;
    int z = 6;
    int w = 14;
    y -= 7;
    Console.WriteLine("y = {0}", y);



      a *= x;
      Console.WriteLine("a = {0}", a);
      x /= y;                                     // Note 2
      Console.WriteLine("x = {0}", x);
      w %= z;
      Console.WriteLine("w = {0}", w);
      b *= z + 3;                                 // Note 3
      Console.WriteLine("b = {0}", b);
    }
  }
```

```
 y = -2
 a = 6
 x = -1
 w = 2
 b = 36
```

```
y = -2
a = 6
x = -1
w = 2
b = 36
```

Note 1: int a = 2;

We initialize all variables

Note 2: x /= y;

The program changed the value of y, so the value of y used here is its current value of -2, rather than its initial value, 5

Note 3: b *= z + 3;

Recall that the equivalent expression is b*(z+3), because the entire right-hand side expression is multiplied by the left-hand side variable, b

Try It Yourself ➤

✓ Test Your Understanding

17. What value would C# assign each variable if, for each expression, j=7, k=11, and n=-4?

a.j += 31;

b.k *= n;

c.k -= n + 7;

d.k %= j

e.k /= n - 1

Try ItYourself➤

18. Change the variable initializations in Example 2.11 to a=7, b=2, x=12, y=4, z=-6, and w=8. What output do you expect from this modified program? Compile and run it to see if your expectations are correct.

Increment and Decrement Operators

C# has simple forms for the frequent operations of adding 1 to a variable (incrementing) or subtracting 1 from a variable (decrementing). To increment the variable

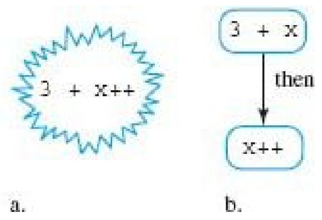


FIGURE 2.12 Expression (a) and equivalent expressions (b)

x using the postfix form of the increment operator, we write x++. If x had a value of 5, executing x++ would give it a value of 6

There is significance to putting the plus signs after the variable (postfix). If you use this postfix increment in an expression, the computation uses the old value of the variable, and then increments it. So if x is 5, evaluating the expression 3 + x++ will give 8, and then the value of x changes to 6. Figure 2.12 shows that evaluating 3 + x++ is like evaluating two expressions, first 3 + x, and then x++

The prefix form of the increment operator, ++x, also increments the variable by 1, but it does it before the variable is used in an expression. If x had a value of 5, evaluating the expression 3 + ++x would increment x to 6 and then evaluate the expression, giving a value of 9. Figure 2.13 shows that evaluating 3 + ++x is like evaluating two expressions, first ++x, and then 3 + x C# has two forms of the decrement operator. The postfix decrement, x--, uses the value of x and then decrements it, so if x is 3, then 2 + x-- evaluates to 5, and x changes to 2. The prefix decrement, --x, decrements x and then uses that new value of x, so if x is 3, then 2 + --x evaluates to 4, because x was decremented to 2 before the expression was evaluated

TIP

We illustrate the use of the increment and decrement operators in expressions such as 3 + x++. For clarity, it is better to limit their use to variables not part of larger expressions. We could replace 3 + x++ with the two statements

```
3 + x;
x++;y
```

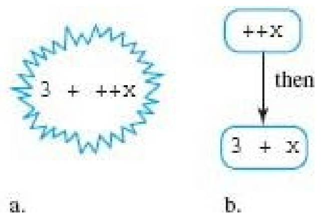


FIGURE 2.13 Expression (a) and equivalent expressions (b)

EXAMPLE 2.12 Increment.cs

```
/* Uses the prefix and postfix increment and decrement
 * operators
 */

using System;
public class Increment {
  public static void Main() {
    int x = 5;
    int y = 5;
    int a = 3;
    int b = 3;
    int j = 7;
    int k = 7;
    int result = 0;
    j++;                                        // Note 1
    ++k;
    Console.WriteLine(”j = {0} and k = {1}”,  j, k);
    j--;
    --k;
    Console.WriteLine(”j = {0} and k = {1}”,  j, k);
    result = 3 +  x++;
    Console.WriteLine(”result = {0} and x = {1}”, result, x);
    result = 3 + ++y;                           // Note 2
    Console.WriteLine(”result = {0} and y = {1}”, result, y);
    result = 2 + a--;
    Console.WriteLine(”result = {0} and a = {1}”, result, a);;
    result = 2 + --b;
    Console.WriteLine(”result = {0} and b = {1}”, result, b);
  }
}
```

```
j = k and 8 = 8
j = k and 7 = 7
result = 8 and x = 6
result = 9 and y = 6
result = 5 and a = 2
result = 4 and b = 2
```

Output

Note 1: `j++;`

We could have put this and the next statement on a single line as in

j++; ++k;

Usually, the better style puts each statement on a separate line

Note 2: result = 3 + ++y;

We leave a space after the first plus sign. Otherwise, if we wrote 3

+++ y, C# would try to evaluate 3++ +y which would give a compiler error, because we can only increment a variable, not a number like 3, which is constant

The BIG Picture

Arithmetic expressions use precedence rules that model those of mathematics. These allow us to use fewer parentheses, making expressions easier to read and write. Like C and C++, C# can combine arithmetic and assignment operators and includes simple increment and decrement operators

Try It Yourself

✓ Test Your Understanding

19. Evaluate each of the following C# expressions, where for each, x=5, y=7.

a.x--

b.y++ + 6

c.y * --x

d.x++ /3

e.++x/3

f.++y + --x

20. Initialize the variables in Example 2.12 to x=7, y=6, a=5, b=-2, j=4, and k=3. What output do you expect from this modified program? Compile and run it to see if your expectations are correct.

2.5 METHODS AND PARAMETERS

So far, our programs have had exactly one method, named Main. The system that executes a compiled C# program looks for the method named Main to start its execution of our program

In order to raise the level of abstraction of our program, we should define additional methods, as we discussed in Section 1.6 on software development. In this section, we show how to define methods and revise some of our earlier examples to use methods. We use class methods here, but will study instance methods when we introduce objects in Chapter 5

Methods

We can create programs with more than one method. A method can contain the code for an operation we need to repeat. The method name serves as the name of a new operation we have defined. As a simple example, let us define a method to multiply a value by four.[5]

Of course, we have to tell our method which value we want to multiply. Let us name this method MultiplyBy4 and name the value aNumber. Our method declaration is

```
public static int MultiplyBy4(int aNumber) {
  return 4*aNumber;
}
```

The modifier static indicates that this is a **class method.** A class method is part of the class in which it is declared.[6] Figure 2.14 shows the class diagram for the Multiply class we will use in Example 2.13

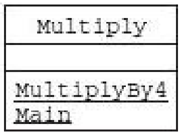
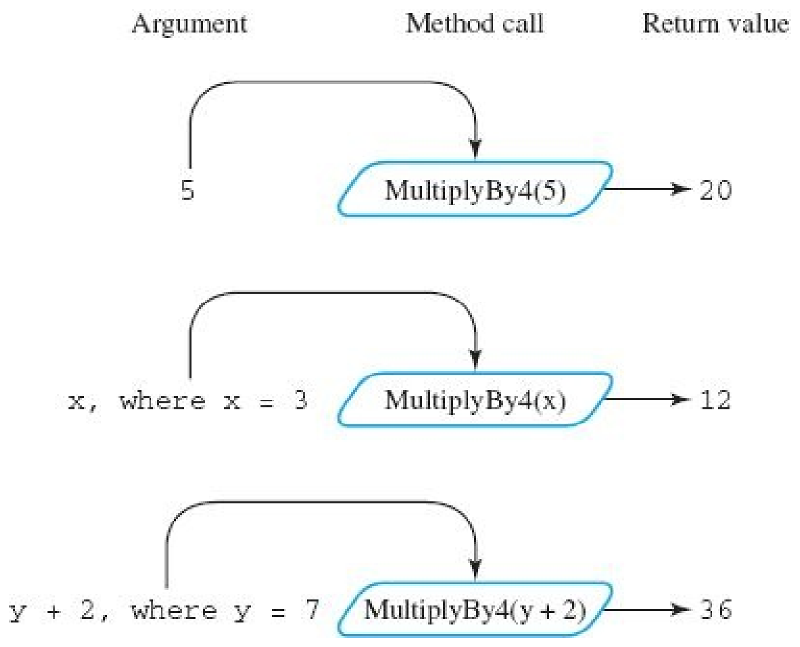


FIGURE 2.14 The Multiply class

We use parameters to communicate with methods and to make them more flexible. The MultiplyBy4 method has one parameter, the integer aNumber. We call the parameter aNumber a formal parameter. It specifies the form of the data we will pass to the method when we call it. Each parameter functions as a local variable inside the method A method can return a value, the result of the operation. We use the return statement to specify the result. Here we return four times the parameter, aNumber, that we pass into the method. Note the type name, int, just before the method name MultiplyBy4. It specifies the type of the result the MultiplyBy4 method returns To use the MultiplyBy4 method we pass it a value of the type specified by the formal parameter, which is an int . For example,



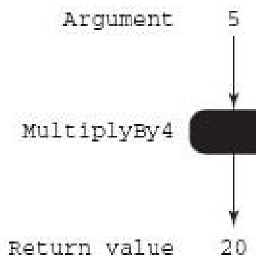
In this text, we use the term argument to denote the actual value passed when we call a method, reserving the term parameter for the formal parameter we use in the definition of the method

We can think of arguments as the raw materials of the method, which is like a machine that uses the raw materials to produce a product, the return value. Figure 2.15 shows this view of the MultiplyBy4 method. We show the method as a black box because we can use it without having to look inside to see how it works

EXAMPLE 2.13 Multiply.cs

```
/*  Defines a multiplyBy4 method and uses it in a program
 */

using System;
public class Multiply {

        // Multiplies its argument by four
  public static int MultiplyBy4(int aNumber) {      // Note 1
    return 4*aNumber;
  }

      // Shows various ways of passing arguments to a method
  public static void Main() {
    int x = 7;
    int y = 20;
    int z = -3;
    int result = 0;
    result = MultiplyBy4(x);                        // Note 2
    Console.WriteLine
            ("Passing a variable, x: {0}", result);
    result = MultiplyBy4(y+2);                      // Note 3
    Console.WriteLine
            ("Passing an expression, y+2: {0}", result);
    result = 5 + MultiplyBy4(z);                    // Note 4
    Console.WriteLine
      ("Using MultiplyBy4 in an expression: {0}", result);
    result = MultiplyBy4(31);                       // Note 5
```

FIGURE 2.15 The MultiplyBy4 "machine"



```
    Console.WriteLine
            ("Passing a constant, 31: {0}", result);
    Console.WriteLine
           ("Passing an expression to WriteLine: {0}",
             MultiplyBy4(y));                       // Note 6
  }
}
```

```
Passing a variable, x: 28
Passing an expression, y+2: 88
Using MultiplyBy4 in an expression: -7
Passing a constant, 31: 124
Passing an expression to WriteLine: 80
```

Note 1: `public static int MultiplyBy4(int aNumber) {`

We declare the method MultiplyBy4 with integer parameter aNumber and integer return value. The body of the method contains the code to implement the operation and compute the return value. Here we compute an expression, 4*aNumber, and return this value. Note the semicolon we use to terminate the return statement

Note 2: `result = MultiplyBy4(x);`

We call the method, passing it an argument x of type int, the same type we specified in the declaration. The MultiplyBy4 method multiplies the value, 7, of x by 4 and returns the value 28

Note 3: `result = MultiplyBy4(y+2);`

We can substitute an expression, y+2, for the parameter. Because y is 20, y+2 is 22, and the return value will be 88

Note 4: `result = 5 + MultiplyBy4(z);`

If a method returns a value, we can use that method in an expression. Here z is —3, so the return value from MultiplyBy4 will be —12 and the result will be -7

Note 5: `result = MultiplyBy4(31);`

The argument we pass to a method can be a constant value. Here we pass 31, so the result is 124

Note 6:
```
Console.WriteLine
     ("Passing an expression to WriteLine: {0}",
       MultiplyBy4(y));
```

The return value does not necessarily need to be saved in a variable. Here it is part of the argument to the WriteLine method. Because y is

20, MultiplyBy4(y) returns 80, so this WriteLine statement will output

```
Passing an expression to WriteLine: 80
```

A method might not have any parameters, and it might not return a value. Example 2.14 shows a method, PrintBlurb, that has no parameters and has no return value; it simply prints a message

EXAMPLE 2.14 NoArgsNoReturn.cs

```
/*  Shows that a method may not have any parameters, and may
 *  not return a value.
 */

using System;
public class NoArgsNoReturn {

      // Displays a message
  public static void PrintBlurb()  {               // Note 1
    Console.WriteLine(“This method has no arguments, ”
                 + “and it has no return value.”); // Note 2
  }                                                // Note 3

          // Execution starts here
  public static void Main( ) {
    PrintBlurb();                                  // Note 4
  }
}
```

```
This method has no arguments, and it has no return value
```

This method has no arguments, and it has no return value

Note 1: `public static void PrintBlurb()  {`

Even when a method has no parameters, we still use the rounded parentheses, but with nothing between them. We use void to show that the method has no return value.

Note 2:
```
Console.WriteLine(“This method has no arguments, ”
           + “and it has no return value.”);
```

When we use the plus sign with string arguments it represents string concatenation, which joins the two strings into one longer string

Note 3: `}`

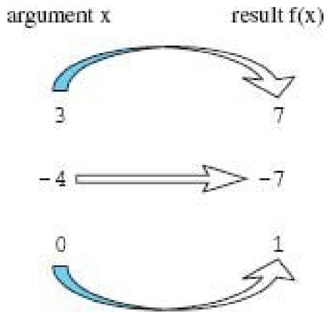
We do not need a return statement because PrintBlurb does not return any value

Note 4: `PrintBlurb();`

When calling a method with no arguments, use the empty parentheses. Because PrintBlurb has no return value, we cannot use it in an expression the way that we did with the MultiplyBy4 method

A Little Extra: **Methods and Functions**

Methods in C# are similar to functions in other languages. In mathematics, a function gives a correspondence between the argument passed to the function and the resulting function value. The function f, given by f(x)=2x+1, computes



values as follows:

We could program a C# method

```
int F(int x) {
  return 2*x + 1;
}
```

which computes the same values, in its range, as the mathematical function f. We call the C# implementation a method instead of a function because we must declare it inside a class. We will cover more about classes later in this text. Other languages use the name function for a similar program that is not declared within a class

Passing by Value

By default, C# passes arguments by value, meaning that the called method receives the value of the argument rather than its location. Figure 2.16 illustrates what happens in Example 2.13 when we call MultiplyBy4(x). Main has a variable x whose value is 7. The MultiplyBy4 method has a parameter aNumber, which

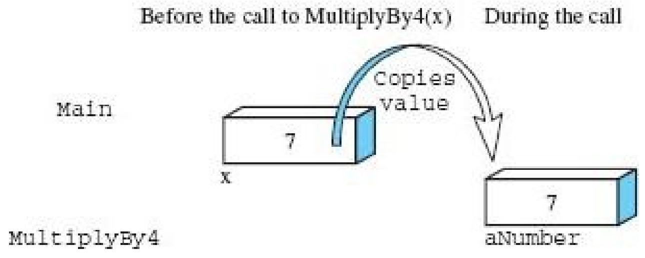


FIGURE 2.16 Passing by value

functions as a local variable. The call MultiplyBy4(x) causes C# to copy the value 7 to the variable aNumber in the MultiplyBy4 method

Example 2.15 illustrates the effect of passing by value. We create a method that returns the cube of the argument passed to it. Passing x, which has a value of 12, will cause the Cube method to return 1728 which equals 12*12*12. We named the parameter to the Cube method, aNumber. Inside the Cube method, we add 5 to the value of aNumber, but this has no effect on the argument x, defined in the Main method, which remains 12

We added a local variable, result, to the Cube method . We declare local variables inside a method. They may be used only inside the method in which they are declared. The **scope** of a variable signifies the region of code in which it is visible. The scope of a local variable is the method in which it is declared. The variables we declared in our previous examples in this chapter are all local variables because we declared them inside the Main method EXAMPLE 2.15 PassByValue.cs

```
/*  Illustrates pass by value
 */

using System;
public class PassByValue {

    // Returns the cube of its argument
  public static int Cube(int aNumber) {
    int result = aNumber*aNumber*aNumber;          // Note 1
    aNumber += 5;                                  // Note 2
    return result;
  }

   /* Shows that the value in the caller
    * does not change.
    */
  public static void Main() {
    int x = 12;
    int value = Cube(x);
    Console.WriteLine("The cube of {0} is {1}",
                                x, value);         // Note 3
  }
}
```

```
The cube of 12 is 1728
```

The cube of 12 is 1728

Note 1: `int result = aNumber*aNumber*aNumber;`

The variable result is local to the cube method and may only be used there.

Note 2: `aNumber += 5;`

We add 5 to aNumber to show that this change affects only aNumber, and not the variable x which we pass to it from Main.

Note 3:
```
Console.WriteLine("The cube of {0} is {1}",
                           x, value);
```

Console.WriteLine("The cube of {0} is {1}", x, value);

When we pass x to the Cube method,C# copies its value,12,to the parameter aNumber, which functions as a local variable of the Cube method. Changing aNumber has no effect on the value of the variable x, which remains 12.

Figure 2.17 illustrates the operations of Example 2.15. We see that local variables and parameters are alive only during the method call. They do not exist before or after the call. We see that C# copies the value of the argument, so the change to the parameter aNumber inside the Cube method has no effect on the value of the argument x in the Main method

Programming with Methods

In Section 1.4 we discussed abstracting operations to make our programs rise above the level of generalpurpose C# constructs. Putting a big block of low-level C# code in a Main method does not present our design in a meaningful way to the reader of the program. Our programming examples in this chapter illustrate basic concepts, and are not meant to provide a useful application. Nevertheless, even here we can introduce some abstraction and organize our code using methods

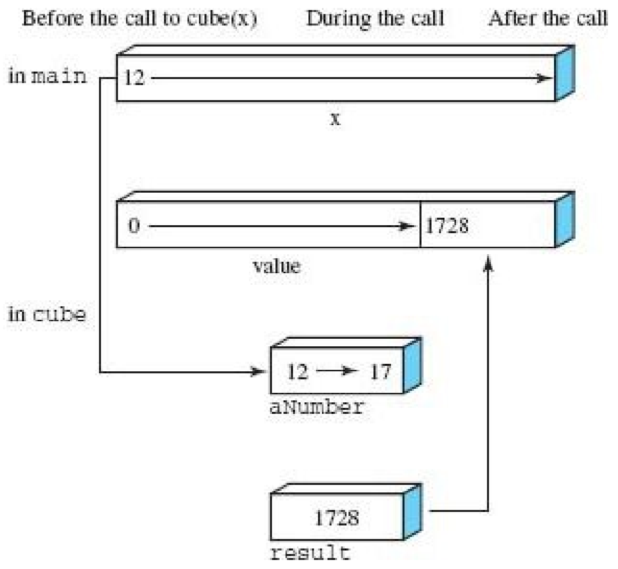
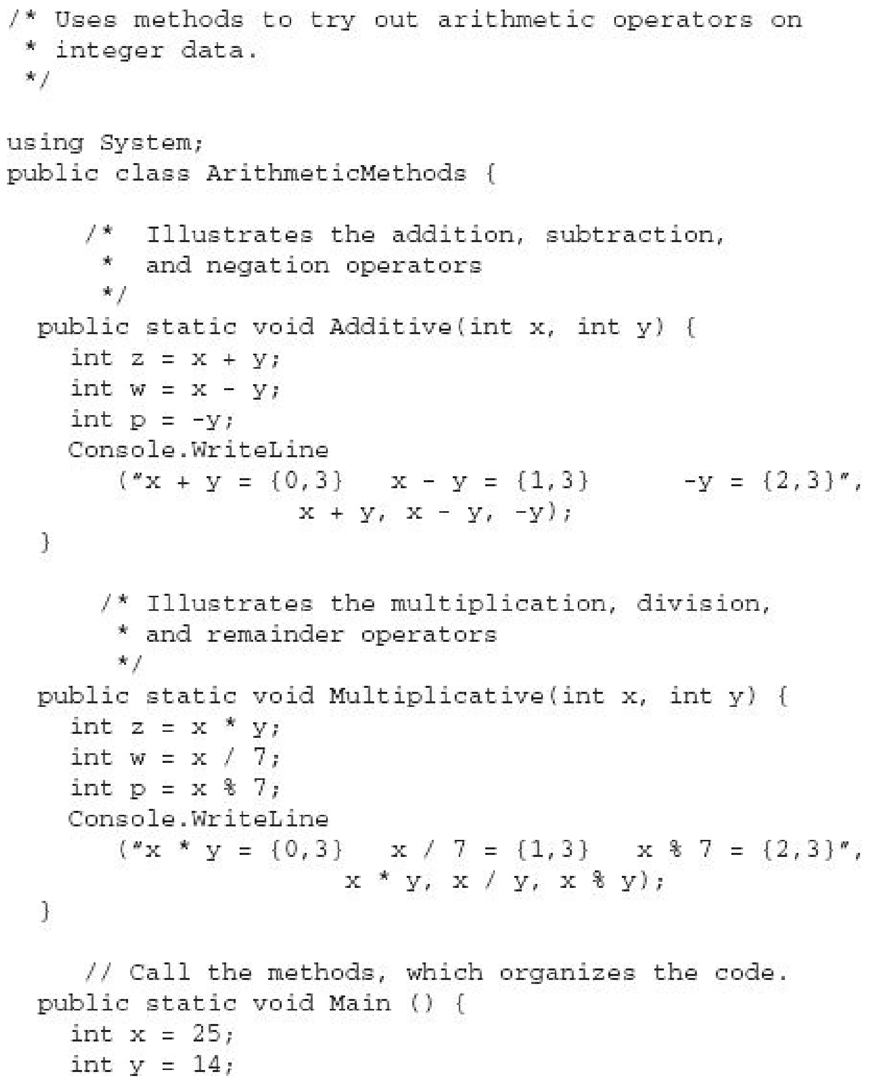


FIGURE 2.17 Memory usage for parameter passing

For example, let us reconsider Example 2.9, Arithmetic.cs. Code in the Main method illustrates C# arithmetic operations. Example 2.16 organizes this code into two methods, called Additive and Multiplicative . Each has two integer parameters. Additive illustrates the +, -, and negation operators, and Multiplicative illustrates *, /, and %. The Main method simply calls each of these methods. It would be easy to call these methods with different data, something that would require us to copy the code again using the approach of Example 2.9

EXAMPLE 2.16 : ArithmeticMethods.cs



```
/* Uses methods to try out arithmetic operators on
 * integer data.
 */

using System;
public class ArithmeticMethods {

     /*  Illustrates the addition, subtraction,
      *  and negation operators
      */
  public static void Additive(int x, int y) {
    int z = x + y;
    int w = x - y;
    int p = -y;
    Console.WriteLine
       ("x + y = {0,3}   x - y = {1,3}      -y = {2,3}",
                   x + y, x - y, -y);
  }

      /* Illustrates the multiplication, division,
       * and remainder operators
       */
  public static void Multiplicative(int x, int y) {
    int z = x * y;
    int w = x / 7;
    int p = x % 7;
    Console.WriteLine
       ("x * y = {0,3}   x / 7 = {1,3}   x % 7 = {2,3}",
                      x * y, x / y, x % y);
  }

     // Call the methods, which organizes the code.
  public static void Main () {
    int x = 25;
    int y = 14;
```

```
      Additive(x, y);
      Multiplicative(x, y);
    }
  }
```

```
x + y =  39   x - y =  11      -y = -14
x * y = 350   x / 7 =   1   x % 7 =  11
```

Output

Named and Optional Arguments

Using named arguments we can pass arguments by name so that we do not not have to remember their order in the method declaration. Optional arguments allow us to specify default values for arguments. If we call and method and do not pass a value for an optional argument it will take the default value For example, in Example 2.16 we could declare the Multiplicative method as

```
public static void Multiplicative(int x = 25, int y = 14)
```

so that x has the default value of 25 and y has the default value of 14. We could then rewrite Main as

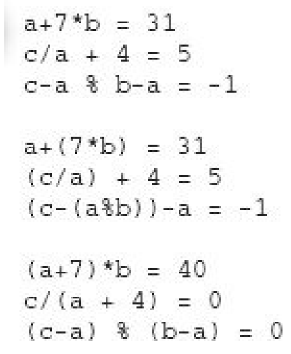
```
public static void Main () {
  Additive(y: 14, x: 25);
  Multiplicative();
}
```

The Additive method has arguments passed by name so that we may pass them in a different order than they appear in the declaration. The Multiplicative method has optional arguments so even though we do not pass any arguments explicitly, x has the default value of 25 and y is 14

Similarly, we can reorganize Example 2.10, Precedence.cs, to abstract meaningful methods. We define a NoParen method to evaluate three expressions without parentheses. A SameParen method evaluates the same expressions, which have parentheses inserted according to the precedence rules. If we insert parentheses correctly, the results from SameParen should be the same as the results from NoParen Finally, the ChangeParen method inserts parentheses to make C# evaluate the expressions in an order different from that following the precedence rules. Results from this method may differ from those obtained from the previous two methods. Again, it would be easy to call the methods with different input

EXAMPLE 2.17 :PrecedenceMethods.cs

```
/* Uses methods to illustrate precedence rules
 * for arithmetic operators.
 */



using System;
public class PrecedenceMethods {

    // Uses no parentheses
  public static void NoParen(int a, int b, int c) {
    int expr1 = a + 7 * b;
    int expr2 = c / a + 4;
    int expr3 = c - a % b - a;
    Console.WriteLine(“a+7*b = {0}”, expr1);
    Console.WriteLine(“c/a + 4 = {0}”, expr2);
    Console.WriteLine(“c-a % b-a = {0}”, expr3);
    Console.WriteLine();
  }

    // Inserts parentheses following precedence rules
  public static void SameParen(int a, int b, int c) {
    int expr1 = a + (7 * b);
    int expr2 = (c / a) + 4;
    int expr3 = (c - (a % b)) - a;
    Console.WriteLine(“a+(7*b) = {0}”, expr1);
    Console.WriteLine(“(c/a) + 4 = {0}”, expr2);
    Console.WriteLine(“(c-(a%b))-a = {0}”, expr3);
    Console.WriteLine();
  }

    // Inserts parentheses to change the normal
    // evaluation order
  public static void ChangeParen(int a, int b, int c) {
    int expr1 = (a + 7) * b;
    int expr2 = c / (a + 4);
    int expr3 = (c - a) % (b - a);
    Console.WriteLine(“(a+7)*b = {0}”, expr1);
    Console.WriteLine(“c/(a + 4) = {0}”, expr2);
    Console.WriteLine(“(c-a) % (b-a) {0}”, expr3);
    Console.WriteLine();
  }

     // Calls each method to compare results
  public static void Main() {
    NoParen(3,4,5);
    SameParen(3,4,5);
    ChangeParen(3,4,5);
  }
}
```



```
a+7*b = 31
c/a + 4 = 5
c-a % b-a = -1

a+(7*b) = 31
(c/a) + 4 = 5
(c-(a%b))-a = -1

(a+7)*b = 40
c/(a + 4) = 0
(c-a) % (b-a) = 0
```

The BIG Picture

A method contains code for an operation. We use parameters to pass values to a method, and may return a value from the method. When we change the value of a parameter inside a method, it has no effect on the value of the variable passed from the caller. We declare local variables inside a method, and can use them only there

Using methods to abstract operations helps to organize code into meaningful units

Try It Yourself ➤

Try It Yourself ➤

✓ Test Your Understanding

21. What value will the method Multiplyby4 return given that x=-11, y=23, and z = 6, and that MultiplyBy4 is called with the following argument?

a.x

b.y-5

c.-5

d.z*x+10

e.y-x

22. Initialize the variables in Example 2.13 to x=-5, y=14, and z = 7. What output do you expect from this modified program? Compile and run it to see if your expectations are correct

23. Change Example 2.13 to use a method Add4 instead of MultiplyBy4 . The method Add4 will add 4 to the parameter aNumber and return that value. Compile and run the new version in which Add4 replaces MultiplyBy4, checking that the results are what you expect.

24. Consider the method declaration

int MyAge Is (int myAge) {return 39;}

a. What is the name of this method?

b. What is the type of its return value?

c. What is the name of its formal parameter?

d. What is the type of its formal parameter?

e. Write a statement that calls the method with the argument 55

SUMMARY

■ To begin writing C# programs, we need to know the basic elements of the C# language We name our data and other items using identifiers, which must start with a letter (or underscore, _), followed by letters or digits or both, and can be of any length. C# is case sensitive, distinguishing between identifiers fruit and Fruit, for example. Keywords such as int are reserved for special uses and cannot be used as identifiers. C# uses the Unicode character set, which contains thousands of characters, including all the commonly used ASCII characters

■ A variable holds data that the program uses. Every C# variable has a type and a name that the programmer must declare. C# uses the keyword int for an integer data type. Declaring an int variable specifies its type as integer. Integer variables can hold values of up to 10 decimal digits. We can initialize a variable in its declaration, which will give that variable an initial or starting value. We use the assignment statement to change a variable's value during the execution of the program

■ To perform computations, C# provides the binary arithmetic operators +, *-, , /, and %, and the unary arithmetic operators + and -. C# uses precedence rules to evaluate arithmetic expressions without having to clutter them with too many parentheses. Multiplication, division, and remainder have higher precedence than addition and subtraction. C# has operators* +=, -=, =, /=, and %= that combine assignment with the other arithmetic operators, and increment and decrement operators, ++ and --, which come in either prefix or postfix forms