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.

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

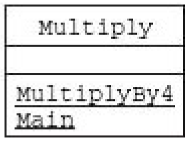
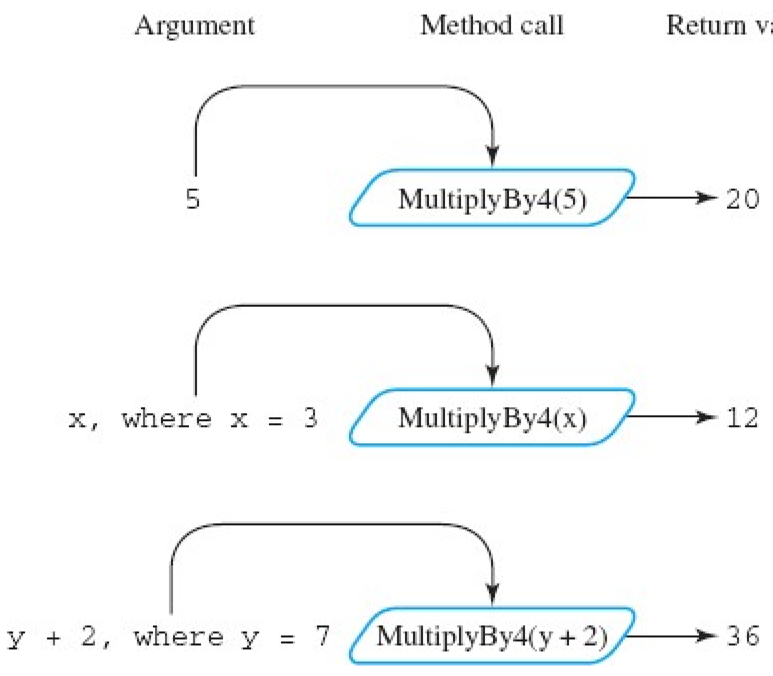


FIGURE 2.14 The Multiply class

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
```

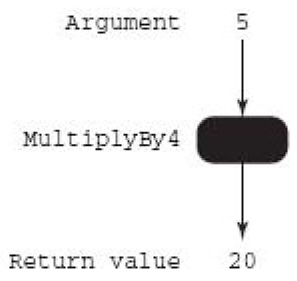


FIGURE 2.15 The MultiplyBy4 "machine"

```
    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
```

Output

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
```

Output

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: `}`

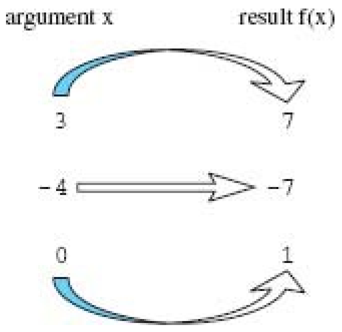
We do not need a return statement because PrintBlurb does not return any value

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

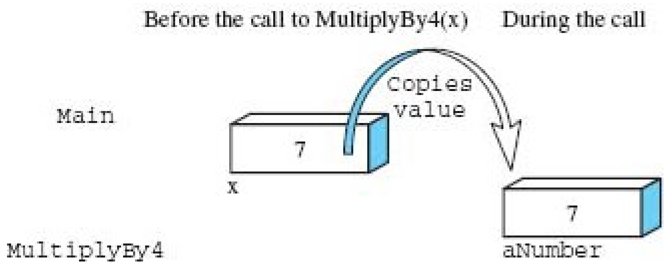


FIGURE 2.16 Passing by value

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
```

Output

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 general-purpose 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

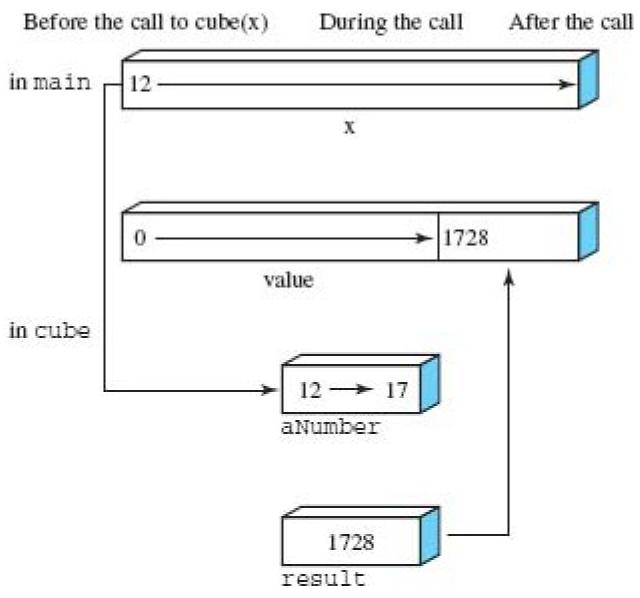


FIGURE 2.17 Memory usage for parameter passing

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

USING THE MATH LIBRARY

The C# language provides the basic arithmetic operations of addition, subtraction, multiplication, and division, but not exponentiation, max, min, and other mathematical functions. As we shall see, C# adds many resources as classes that we can use in our programs.

In this section we introduce methods of the Math class to compute square root, absolute value, powers, max, min, random numbers, and other mathematical functions which we can use to solve problems in the remainder of the text. To use these methods, we prefix the method name with the class name, Math, as in Math.Sqrt(2.0), which computes the square root of 2.0.

When calling class methods (methods that use the static modifier) from a method of the same class, as in Example 4.4, where we call Sum and Max from Main, we do not need to use the class prefix. Optionally, we could use the class prefix, calling the Sum method as UserMenu.Sum(). When calling class methods from a method of a different class, as in Example 4.8, where we call IO.GetInt, we must use the class prefix.

Powers and Roots

The power method, Pow, takes two arguments of type double and returns a double value, which is the result of raising the first argument to the exponent given by the second argument. Some examples follow:

```
Math.Pow(2.0,3.0)  returns 8.0   (2.0$^{3.0}$)
Math.Pow(3.0,2.0)  returns 9.0   (3.0$^{2.0}$)
```

The square root method, Sqrt, takes a double argument and returns a double value, the square root of its argument. Some examples are

```
Math.Sqrt(2.0)    returns 1.4142135623730951
Math.Sqrt(16.0)   returns 4.0
```

Maximum, Minimum, and Absolute Value

C# provides Max, Min, and Abs methods for each of the numeric types, including int, long, float, and double. For each type, the Max method returns the maximum of its two arguments of that type, and the Min method returns the minimum of its two arguments. Some examples are

```
Math.Max(3, 4)            returns 4
Math.Max(17.32, 5.8567)   returns 17.32
Math.Max(-9, -11)         returns -9
Math.Min(3, 4)            returns 3
Math.Min(17.32, 5.8567)   returns 5.8567
Math.Min(-9, -11)         returns -11
```

For each type, the Abs method returns the absolute value of its argument, which, by definition, is

```
Math.Abs(x) =  x, if x >= 0
            = -x, if x <  0
```

Informally, the Abs method removes the minus sign, if any. Some examples are

```
Math.Abs(-10)     returns 10
Math.Abs(12.34)   returns 12.34
```

Floor and Ceiling

Informally, just as the absolute value removes the minus sign, the floor removes the fractional part of its double argument, returning the largest double value that is not greater than the argument and is equal to a mathematical integer. Some examples follow:

```
Math.Floor(25.194)  returns 25.0
Math.Floor(-134.28) returns -135.0
```

The ceiling also removes the fractional part, but it returns the smallest double value that is not less than the argument and is equal to a mathematical integer. Some examples include

```
Math.Ceiling(25.194)    returns 26.0
Math.Ceiling(-134.28)   returns -134.0
```

Pi and E

In addition to methods, the Math class contain two important constants, Math.PI, the circumference of a circle of diameter one, and Math.E, the base for natural logarithms. We will use Math.PI in our calculations of areas and volumes.

EXAMPLE 4.9 Library.cs

```
/* Uses methods from the Math class.
 */

using System;
public class Library {
  public static void Main()  {
     Console.WriteLine('Two cubed is {0}", Math.Pow(2.0,3.0));
     Console.WriteLine('Three squared is {0}',
                                      Math.Pow(3.0,2.0));
     Console.WriteLine('The square root of two is {0}",
                                      Math.Sqrt(2.0));
     Console.WriteLine('The square root of 16 is {0}',
                                      Math.Sqrt(16.0));
     Console.WriteLine('The max of 2.56 and 7.91/3.1 is {0}",
     Math.Max(2.65,7.91/3.1));                         // Note 1
     Console.WriteLine('The min of -9 and -11 is {0}',
                           Math.Min(-9,-11));
     Console.WriteLine('The absolute value of -32.47 is {0}",
                           Math.Abs(-32.47));
     Console.WriteLine('The floor of 25.194 is {0}',
                           Math.Floor(25.194));
     Console.WriteLine('The ceiling of 25.194 is {0}',
                           Math.Ceiling(25.194));
     Console.WriteLine('Pi is {0}", Math.PI);
     Console.WriteLine
        ("Pi, to six decimal places, is {0:F6}", Math.PI);
                                                       // Note 2

            Console.WriteLine
               ("The base of natural logarithms, e, is {0}", Math.E);
          }
        }
```

```
Two cubed is 8.0
Three squared is 9.0
The square root of two is 1.4142135623730951
The square root of 16 is 4.0
The max of 2.56 and 7.91/3.1 is 2.65
The min of -9 and -11 is -11
The absolute value of -32.47 is 32.47
The floor of 25.194 is 25
The ceiling of 25.194 is 26
Pi is 3.14159265358979
Pi, to six decimal places, is 3.141593
The base of natural logarithms, e, is 2.71828182845905
```

Note 1: `Math.Max(2.65,7.91/3.1)`

The argument can be an expression that evaluates to a double, as here for the second argument of Max.

Note 2:
```
Console.WriteLine
('Pi, to six decimal places, is {0:F6}', Math.PI);
```

We format the output to show only the first six decimal places of pi.

A Little Extra: Math Functions

The Math class has methods to compute the trigonometric functions, sine, cosine, and tangent. These methods each take a double argument representing an angle in radians, and produce a double result. Some examples are

```
Math.Sin(Math.PI/6.0) returns  0.5
Math.Cos(Math.PI)     returns -1
Math.Tan(Math.PI/4.0) returns  1
```

The exponential function computes e^x, where *e* is the base of natural logarithms. The exp method of the Math class computes the exponential function. The log method computes the natural logarithm function, log x. Some examples include

```
Math.Exp(1.0)    returns 2.71828182845905
Math.Exp(2.0)    returns 7.38905609893065
Math.Log(Math.E) returns 1
Math.Log(10.0)   returns 2.30258509299405
```