CHAPTER THREE

METHODS

3.1 Introduction

In this chapter we study methods. We emphasize how to declare and use methods to facilitate the design, implementation, operation and maintenance of large applications. You'll learn how to declare a method with more than one parameter. You'll also learn how value-type and reference-type arguments are passed to methods, how local variables of methods are maintained in memory and how a method knows where to return after it completes execution.

You'll use or create developing applications will have more than one method of the same name. This technique, called method overloading, is used to implement methods that perform similar tasks but with different types and/or different numbers of arguments.

3.2 static Methods, static Variables and Class Math

Although most methods execute on specific objects in response to method calls, this is not always the case. Sometimes a method performs a task that does not depend on the contents of any object. Such a method applies to the class in which it's declared as a whole and is known as a static method. It's not uncommon for a class to contain a group of static methods to perform common tasks. For example, recall that we used static method ***Pow*** *of class* ***Math*** *to raise a value to a power to declare a method as static, place the keyword static before the return type in the method's declaration. You call any static method by specifying the name of the class in which the method is declared,*

followed by the member access (.) operator and the method name, as in

```
ClassName.MethodName( arguments )
```

We use various methods of the Math class here to present the concept of static methods. Class Math (from the System namespace) provides a collection of methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of 900.0 with the static method call

```
Math.Sqrt( 900.0 )
```

The preceding expression evaluates to 30.0. Method ***Sqrt*** *takes an argument of type double and returns a result of type double. To output the value of the preceding method call in the console window, you might write the statement*

```
Console.WriteLine( Math.Sqrt( 900.0 ) );
```

In this statement, the value that ***Sqrt*** *returns becomes the argument to method* ***WriteLine****. We did not create a Math object before calling method* ***Sqrt****. Also all of Math's methods are static—therefore, each is called by preceding the name of the method with the class name* ***Math*** *and the member access (.) operator. Similarly, Console method* ***WriteLine*** *is a static method of class Console, so we invoke the method by preceding its name with the class name Console and the member access (.) operator. Method arguments may be constants, variables or expressions. If c = 13.0, d = 3.0 and f = 4.0, then the statement*

```
Console.WriteLine( Math.Sqrt( c + d * f ) );
```

*calculates and displays the square root of 13.0 + 3.0 * 4.0 = 25.0—namely, 5.0. Figure 3.1 summarizes several Math class methods. In the figure, x and y are of type double.*

Method	Description	Example
`Abs( x )`	absolute value of x	`Abs( 23.7 )` is 23.7 `Abs( 0.0 )` is 0.0 `Abs( -23.7 )` is 23.7
`Ceiling( x )`	rounds x to the smallest integer not less than x	`Ceiling( 9.2 )` is 10.0 `Ceiling( -9.8 )` is -9.0
`Cos( x )`	trigonometric cosine of x (x in radians)	`Cos( 0.0 )` is 1.0
`Exp( x )`	exponential method e^x	`Exp( 1.0 )` is 2.71828 `Exp( 2.0 )` is 7.38906
`Floor( x )`	rounds x to the largest integer not greater than x	`Floor( 9.2 )` is 9.0 `Floor( -9.8 )` is -10.0
`Log( x )`	natural logarithm of x (base e)	`Log( Math.E )` is 1.0 `Log( Math.E * Math.E )` is 2.0
`Max( x, y )`	larger value of x and y	`Max( 2.3, 12.7 )` is 12.7 `Max( -2.3, -12.7 )` is -2.3
`Min( x, y )`	smaller value of x and y	`Min( 2.3, 12.7 )` is 2.3 `Min( -2.3, -12.7 )` is -12.7
`Pow( x, y )`	x raised to the power y (i.e., x^y)	`Pow( 2.0, 7.0 )` is 128.0 `Pow( 9.0, 0.5 )` is 3.0
`Sin( x )`	trigonometric sine of x (x in radians)	`Sin( 0.0 )` is 0.0
`Sqrt( x )`	square root of x	`Sqrt( 900.0 )` is 30.0
`Tan( x )`	trigonometric tangent of x (x in radians)	`Tan( 0.0 )` is 0.0

Fig. 3.1 | `Math` class methods.

Math Class Constants PI and E

Class Math also declares two static constants that represent commonly used mathematical values: ***Math.PI*** *and* ***Math.E****. The constant* ***Math.PI*** *(3.14159265358979323846) is the ratio of a circle's circumference to its diameter.*

The constant Math.E (2.7182818284590452354) is the base value for natural logarithms (calculated with static Math method Log). These constants are declared in class Math with the modifiers public and const. Making them public allows other programmers to use these variables in their own classes. A constant is declared with the keyword const—its value cannot be changed after the constant is declared. Both PI and E are declared const because their values never change.

3.3 Declaring Methods with Multiple Parameters

Figure 3.2 uses a user-defined method called Maximum to determine and return the largest of three double values that are input by the user. Lines 11–15 prompt the user to enter three double values and read them from the user. Line 18 calls method Maximum to determine the largest of the three double values passed as arguments to the method. When method Maximum returns the result to line 18, the application assigns Maximum's return value to local variable result. Then line 21 outputs result. At the end of this section, we'll discuss the use of operator + in line 21.

```
1  // Fig. 3.2: MaximumFinder.cs
2  // User-defined method Maximum.
3  using System;
4
5  public class MaximumFinder
6  {
7   // obtain three floating-point values and determine maximum value
8     public static void Main( string[] args )
9     {
10       // prompt for and input three floating-point values
11       Console.WriteLine( "Enter three floating-point values,\n"
12          + "  pressing 'Enter' after each one: " );
13       double number1 = Convert.ToDouble( Console.ReadLine() );
14       double number2 = Convert.ToDouble( Console.ReadLine() );
15       double number3 = Convert.ToDouble( Console.ReadLine() );
16
17       // determine the maximum value
18       double result = Maximum( number1, number2, number3 );
19
20       // display maximum value
21        Console.WriteLine( "Maximum is: " + result );
22    } // end Main
23
24    // returns the maximum of its three double parameters
25    public static double Maximum( double x, double y, double z )
26    {
27       double maximumValue = x; // assume x is the largest to start
28
29       // determine whether y is greater than maximumValue
30       if ( y > maximumValue )
31          maximumValue = y;
32
33       // determine whether z is greater than maximumValue
34       if ( z > maximumValue )
35          maximumValue = z;
36
37       return maximumValue;
38    } // end method Maximum
39 } // end class MaximumFinder
```

Fig. 3.2 | User-defined method `Maximum`.

Method Maximum

Consider the declaration of method Maximum (lines 25–38). Line 25 indicates that the method returns a double value, that the method's name is Maximum and that the method requires three double parameters (x, y and z) to accomplish its task. When a method has more than one parameter, the parameters are specified as a comma-separated list. When Maximum is called in line 18, the parameter x is initialized with the value of the argument number1, the parameter y is initialized with the value of the argument number2 and the parameter z is initialized with the value of the argument number3. There must be one argument in the method call for each required parameter (sometimes called a formal parameter) in the method declaration. Also, each argument must be consistent with the type of the corresponding parameter. For example, a parameter of type double can receive values like 7.35 (a double), 22 (an int) or –0.03456 (a double), but not strings like "hello". Section 3.5 discusses the argument types that can be provided in a method call for each parameter of a simple type.

To determine the maximum value, we begin with the assumption that parameter x contains the largest value, so line 27 declares local variable maximumValue and initializes it with the value of parameter x. Of course, it's possible that parameter y or z contains the largest value, so we must compare each of these values with maximumValue. The if statement at lines 30–31 determines whether y is greater than maximumValue. If so, line 31 assigns y to maximumValue. The if statement at lines 34–35 determines whether z is greater than maximumValue. If so, line 35 assigns z to maximumValue. At this point, the largest of the three values resides in maximumValue, so line 37 returns that value to line 18.

When program control returns to the point in the application where Maximum was called, Maximum's parameters x, y and z are no longer accessible. Methods can return at most one value.

Implementing Method Maximum by Reusing Method Math.Max

Recall from Fig. 3.1 that class Math has a Max method that can determine the larger of two values. The entire body of our maximum method could also be implemented with nested calls to Math.Max, as follows:

```
return Math.Max( x, Math.Max( y, z ) );
```

The leftmost call to Math.Max specifies arguments x and Math.Max(y, z). Before any method can be called, all its arguments must be evaluated to determine their values. If an argument is a method call, the method call must be performed to determine its return value. So, in the preceding statement, Math.Max(y, z) is evaluated first to determine the maximum of y and z. Then the result is passed as the second argument to the other call to Math.Max, which returns the larger of its two arguments. Using Math.Max in this manner is a good example of software reuse—we find the largest of three values by reusing Math.Max, which finds the larger of two values.

Assembling Strings with String Concatenation

C# allows string objects to be created by assembling smaller strings into larger strings using operator + (or the compound assignment operator +=). This is known as string concatenation. When both operands of operator + are string objects, operator + creates a new string object in which a copy of the characters of the right operand is placed at the end of a copy of the characters in the left operand. For example, the expression "hello " + "there" creates the string "hello there" without disturbing the original strings.

In line 21 of Fig. 3.2, the expression "Maximum is: " + result uses operator + with operands of types string and double. Every value of a simple type in C# has a string representation. When one of the + operator's operands is a string, the other is implicitly converted to a string, then the two are concatenated. In line 21, the double value is implicitly converted to its string representation and placed at the end of the string "Maximum is: ". If there are any trailing zeros in a double

value, these will be discarded when the number is converted to a string. Thus, the number 9.3500 would be represented as 9.35 in the resulting string.

For values of simple types used in string concatenation, the values are converted to strings. If a bool is concatenated with a string, the bool is converted to the string "True" or "False" (note that each is capitalized). All objects have a ToString method that returns a string representation of the object. When an object is concatenated with a string, the object's ToString method is implicitly called to obtain the string representation of the object. If the object is null, an empty string is written.Line 21 of Fig. 3.2 could also be written using string formatting as

```
Console.WriteLine( "Maximum is: {O}", result );
```

As with string concatenation, using a format item to substitute an object into a string implicitly calls the object's ToString method to obtain the object's string representation.

3.4 Notes on Declaring and Using Methods

You've seen three ways to call a method:

1. *Using a method name by itself to call a method of the same class—such as Maximum(number1, number2, number3) .*
2. *Using a variable that contains a reference to an object, followed by the member access (.) operator and the method name to call a non-static method of the referenced object*
3. *Using the class name and the member access (.) operator to call a static method of a class—such as* ***Convert.ToDouble(Console.ReadLine())*** *in lines 13–15 of Fig. 3.2 or* ***Math.Sqrt****(900.0).*

A static method can call only other static methods of the same class directly (i.e., using the method name by itself) and can manipulate only static variables in the same class directly. To access the class's non-static members, a static method must use a reference to an object of the

class. Recall that static methods relate to a class as a whole, whereas non-static methods are associated with a specific instance (object) of the class and may manipulate the instance variables of that object. Many objects of a class, each with its own copies of the instance variables, may exist at the same time. Suppose a static method were to invoke a non-static method directly. How would the method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the non-static method was invoked? Thus, C# does not allow a static method to access non-static members of the same class directly. There are three ways to return control to the statement that calls a method. If the method does not return a result, control returns when the program flow reaches the method-ending right brace or when the statement

```
return;
```

is executed. If the method returns a result, the statement

```
return expression;
```

evaluates the expression, then returns the result (and control) to the caller.

3.5 Argument Promotion and Casting

Another important feature of method calls is argument promotion—implicitly converting an argument's value to the type that the method expects to receive in its corresponding parameter. For example, an application can call Math method ***Sqrt*** *with an integer argument even though the method expects to receive a double argument.*

The statement

```
Console.WriteLine( Math.Sqrt( 4 ) );
```

correctly evaluates Math.Sqrt(4) and displays the value 2.0. Sqrt's parameter list causes C# to convert the int value 4 to the double value 4.0 before passing the value to Sqrt. Such conversions may lead to compilation errors if C#'s promotion rules are not satisfied. The promotion rules specify which conversions are allowed—that is, which conversions can be performed without losing data. In the Sqrt example

above, an int is converted to a double without changing its value. However, converting a double to an int truncates the fractional part of the double value—thus, part of the value is lost. Also, double variables can hold values much larger (and much smaller) than int variables, so assigning a double to an int can cause a loss of information when the double value doesn't fit in the int. Converting large integer types to small integer types (e.g., long to int) can also result in changed values.

The promotion rules apply to expressions containing values of two or more simple types and to simple-type values passed as arguments to methods. Each value is promoted to the appropriate type in the expression. Figure 3.3 lists the simple types alphabetically and the types to which each can be promoted. Note that values of all simple types can also be implicitly converted to type object.

By default, C# does not allow you to implicitly convert values between simple types if the target type cannot represent the value of the original type (e.g., the int value 2000000 cannot be represented as a short, and any floating-point number with digits after its decimal point cannot be represented in an integer type such as long, int or short). Therefore, to prevent a compilation error in cases where information may be lost due to an implicit

Type	Conversion types
bool	no possible implicit conversions to other simple types
byte	ushort, short, uint, int, ulong, long, decimal, float or double
char	ushort, int, uint, long, ulong, decimal, float or double
decimal	no possible implicit conversions to other simple types
double	no possible implicit conversions to other simple types
float	double
int	long, decimal, float or double
long	decimal, float or double
sbyte	short, int, long, decimal, float or double
short	int, long, decimal, float or double
uint	ulong, long, decimal, float or double
ulong	decimal, float or double
ushort	uint, int, ulong, long, decimal, float or double

Fig. 3.3 | Implicit conversions between simple types.

conversion between simple types, the compiler requires you to use a cast operator to explicitly force the conversion. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause loss of information, but for my purposes here, that's fine." Suppose you create a method Square that calculates the square of an integer and thus requires an int argument. To call Square with a double argument named doubleValue, you would write Square((int) doubleValue). This method call explicitly casts (converts) the value of doubleValue to an integer for use in method Square. Thus, if doubleValue's value is 4.5, the method receives the value 4 and returns 16, not 20.25 (which does, unfortunately, result in the loss of information).

3.6 The .NET Framework Class Library

Many predefined classes are grouped into categories of related classes called namespaces. Together, these namespaces are referred to as the .NET Framework Class Library. Throughout the text, using directives allow us to use library classes from the .NET Framework Class Library without specifying their fully qualified names.
For example, an application includes the declaration

```
using System;
```

in order to use the class names from the System namespace without fully qualifying their names. This allows you to use the unqualified class name Console, rather than the fully qualified class name System.Console, in your code. A great strength of C# is the large number of classes in the namespaces of the .NET Framework Class Library. Some key .NET Framework Class Library namespaces are described in Fig. 3.4, which represents only a small portion of the reusable classes in the .NET Framework Class Library.

Namespace	Description
`System.Windows.Forms`	Contains the classes required to create and manipulate GUIs. (Various classes in this namespace are discussed in Chapters 14–15.)
`System.Windows.Controls` System.Windows.Input System.Windows.Media System.Windows.Shapes	Contain the classes of the Windows Presentation Foundation for GUIs, 2-D and 3-D graphics, multimedia and animation. (You'll learn more about these namespaces in Chapter 24, GUI with Windows Presentation Foundation, Chapter 25, WPF Graphics and Multimedia and Chapter 29, Silverlight and Rich Internet Applications.)
`System.Linq`	Contains the classes that support Language Integrated Query (LINQ). (You'll learn more about this namespace in Chapter 9, Introduction to LINQ and the `List` Collection, and several other chapters throughout the book.)
`System.Data` `System.Data.Linq`	Contain the classes for manipulating data in databases (i.e., organized collections of data), including support for LINQ to SQL. (You'll learn more about these namespaces in Chapter 18, Databases and LINQ.)
`System.IO`	Contains the classes that enable programs to input and output data. (You'll learn more about this namespace in Chapter 17, Files and Streams.)
`System.Web`	Contains the classes used for creating and maintaining web applications, which are accessible over the Internet. (You'll learn more about this namespace in Chapter 19, Web App Development with ASP.NET and Chapter 27, Web App Development with ASP.NET: A Deeper Look.)
`System.Xml.Linq`	Contains the classes that support Language Integrated Query (LINQ) for XML documents. (You'll learn more about this namespace in Chapter 26, XML and LINQ to XML, and several other chapters throughout the book.)
`System.Xml`	Contains the classes for creating and manipulating XML data. Data can be read from or written to XML files. (You'll learn more about this namespace in Chapter 26.)
`System.Collections` `System.Collections.Generic`	Contain the classes that define data structures for maintaining collections of data. (You'll learn more about these namespaces in Chapter 23, Collections.)
`System.Text`	Contains the classes that enable programs to manipulate characters and strings. (You'll learn more about this namespace in Chapter 16, Strings and Characters.)

Fig. 3.4 | .NET Framework Class Library namespaces (a subset).