

# Using Methods

In this chapter you will:

- ◎ Learn about methods and implementation hiding
- ◎ Write methods with no parameters and no return value
- ◎ Write methods that require a single argument
- ◎ Write methods that require multiple arguments
- ◎ Write a method that returns a value
- ◎ Pass array values to a method
- ◎ Learn some alternate ways to write a `Main()` method header
- ◎ Learn about issues using methods in GUI programs

In the first chapters of this book, you learned to create C# programs containing `Main()` methods that declare variables, accept input, perform arithmetic, and produce output. You learned to add decisions, loops, and arrays to your programs. As your programs grow in complexity, their `Main()` methods will contain many additional statements. Rather than creating increasingly long `Main()` methods, most programmers prefer to modularize their programs, placing instructions in smaller “packages” called methods. In this chapter, you learn to create many types of C# methods. You will understand how to send data to these methods and receive information back from them.

## Understanding Methods and Implementation Hiding

A **method** is an encapsulated series of statements that carry out a task. Any class can contain an unlimited number of methods. So far, you have written console-based applications that contain a `Main()` method but no others. You also have created GUI applications with a `Click()` method. Frequently, the methods you have written have **invoked**, or **called**, other methods; that is, your program used a method’s name and the method executed to perform a task for the class. For example, you have created many programs that call the `WriteLine()` and `ReadLine()` methods, and in Chapter 6 you used `BinarySearch()`, `Sort()`, and `Reverse()` methods. These methods are prewritten; you only had to call them to have them work.

For example, consider the simple `HelloClass` program shown in Figure 7-1. The `Main()` method contains a statement that calls the `WriteLine()` method. You can identify method names in a program because they always are followed by a set of parentheses. Depending on the method, there might be an argument within the parentheses. You first encountered the term *argument* in Chapter 1. An argument is the data that appears between the parentheses in a method call. The call to the `WriteLine()` method within the `HelloClass` program in Figure 7-1 contains the string argument “Hello”. Some methods you can invoke don’t require any arguments.



Methods are similar to the procedures, functions, and subroutines used in other programming languages.

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

**Figure 7-1** The `HelloClass` program

In the `HelloClass` program in Figure 7-1, `Main()` is a **calling method**—one that calls another. The `WriteLine()` method is a **called method**.

## Understanding Implementation Hiding

When you call the `WriteLine()` method within the `HelloClass` program in Figure 7-1, you use a method that has already been created for you. Because the creators of `C#` knew you would often want to write a message to the output screen, they created a method you could call to accomplish that task. This method takes care of all the hardware details of producing a message on the output device; you simply call the method and pass the desired message to it. The `WriteLine()` method provides an example of **implementation hiding**, which means keeping the details of a method's operations hidden. For example, when you make a dental appointment, you do not need to know how the appointment is actually recorded at the dental office—perhaps it is written in a book, marked on a large chalkboard, or entered into a computerized database. The implementation details are of no concern to you as a client, and if the dental office changes its methods from one year to the next, the change does not affect your use of the appointment-making method. Your only concern is the way you interact with the dental office, not how the office tracks appointments. In the same way that you are a client of the dental practice, a method that uses another is a **client** of that method.

Hidden implementation methods often are said to exist in a black box. A **black box** is any device you can use without knowing how it works internally. The same is true with well-written program methods; the invoking program or method must know the name of the method it is using and what type of information to send it, but the program does not need to know how the method works. Later, you might substitute a new, improved method for the old one, and if the user's means of accessing the method does not change, you won't need to make any changes in programs that invoke the method. The creators of `C#` were able to anticipate many of the methods that would be necessary for your programs; you will continue to use many of these methods throughout this book. However, your programs often will require custom methods that the creators of `C#` could not have expected. In this chapter, you will learn to write your own custom methods.



To more easily incorporate methods into a program, it is common practice to store methods (or groups of associated methods) in their own classes and files. Then you can add the methods into any application that uses them. The resulting compound program is called a **multifile assembly**. As you learn more about `C#`, you might prefer to take this approach with your own programs. For simplicity, most example methods in this book are contained in the same file as any other methods that use them.

### TWO TRUTHS & A LIE

#### Understanding Methods and Implementation Hiding

1. A method is an encapsulated series of statements that carry out a task.
2. Any class can contain an unlimited number of methods.
3. All the methods that will be used by your programs have been written for you and stored in files.

The false statement is #3. As you write programs, you will want to write many of your own custom methods.

## Writing Methods with No Parameters and No Return Value

The output of the program in Figure 7-1 is simply the word *Hello*. Suppose you want to add three more lines of output to display a standard welcoming message when users execute your program. Of course, you can add three new `WriteLine()` statements to the existing program, but you also can create a method to display the three new lines.

Creating a method instead of adding three lines to the existing program is useful for two major reasons:

- If you add a method call instead of three new lines, the `Main()` method will remain short and easy to follow. The `Main()` method will contain just one new statement that calls a method rather than three separate `WriteLine()` statements.
- More importantly, a method is easily *reusable*. After you create the welcoming method, you can use it in any program, and you can allow other programmers to use it in their programs. In other words, you do the work once, and then you can use the method many times.



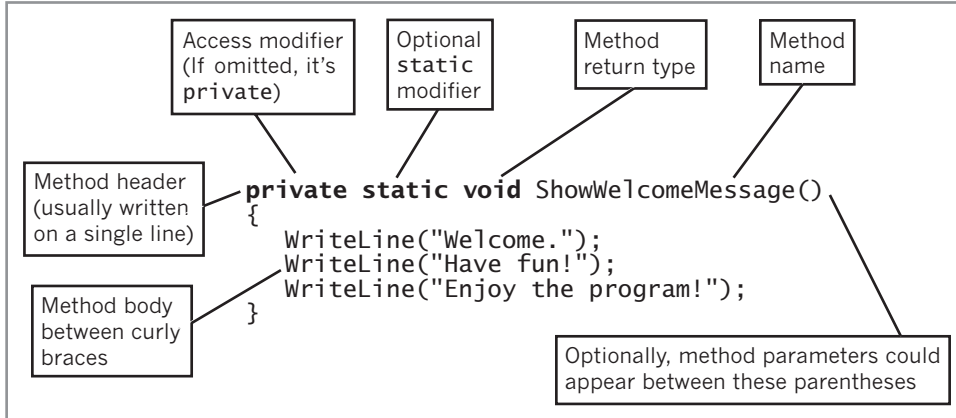
When you place code in a callable method instead of repeating the same code at several points in a program, you are avoiding **code bloat**—a colorful term that describes unnecessarily long or repetitive statements.

Figure 7-2 shows the parts of a C# method. A method must include:

- A **method declaration**, which is also known as a **method header** or **method definition**
- An opening curly brace
- A method body, which is a block of statements that carry out the method's work
- A closing curly brace

The method declaration defines the rules for using the method. It contains:

- Optional declared accessibility
- An optional `static` modifier
- The return type for the method
- The method name, or identifier
- An opening parenthesis
- An optional list of method parameters (separated with commas if there is more than one parameter)
- A closing parenthesis



**Figure 7-2** The ShowWelcomeMessage() method

## An Introduction to Accessibility

The optional declared **accessibility** for a method sets limits as to how other methods can use your method; accessibility can be any of the levels described in Table 7-1. The two levels you will use most frequently, `public` and `private` access, are shaded in the table. You will learn about protected access and what it means to derive types in the chapter “Introduction to Inheritance.”

- **Public access** is established by including a `public` modifier in the member declaration. This modifier allows access to the method from other classes.
- **Private access** is established by including a `private` modifier in the member declaration or by omitting any accessibility modifier. This modifier limits method access to the class that contains the method.

Declared accessibility	Can methods contained in the same class access this method?	Can derived classes access this method?	Can assemblies or projects that contain this class access this method?	Can any class access this method?
<code>public</code>	Yes	Yes	Yes	Yes
<code>protected internal</code>	Yes	Yes	Yes	No
<code>protected</code>	Yes	Yes	No	No
<code>internal</code>	Yes	No	Yes	No
<code>private</code>	Yes	No	No	No

**Table 7-1** Summary of method accessibility

If you do not provide an accessibility modifier for a method, it is `private` by default. The `Main()` method headers that you have seen in all of the examples in this book have no access modifier, so they are `private`. You could explicitly add the keyword `private` to the method headers in any of the programs you have seen so far in this book, and you would notice no difference in execution: The methods still would be `private`. Actually, you could also substitute the `public` keyword in the `Main()` method headers without noticing any execution differences either—it's just that with that modification, other classes could have called the public `Main()` methods by using the appropriate class name and a dot before the method name. When you use a method's complete name, including its class, you are using its **fully qualified** name. In Chapter 1, you learned that you can use the fully qualified method call `Console.WriteLine()` to produce output. When you include either `using System;` or `using static System.Console;` at the top of a program, then you do not need to use the fully qualified method name. Either way, because the classes you write can use the `Console` class method `WriteLine()`, you know the method must not be `private`.

In Chapter 3, you learned to create simple GUI applications, and you learned how to automatically generate a `Click()` method for a button; the generated method header started with the keyword `private`. For example, a private method named `button1_Click()` cannot be used by any other class. If you changed the method's access specifier to `public`, it could be used by another class, but you probably would have no reason to do so. As you study C#, you will learn how to decide which access modifier to choose for each method you write. For now, unless you want a method to be accessible by outside classes, you should use the `private` access modifier with the method. This book uses no modifiers with `Main()` methods because Visual Studio does not create them in its automatically generated code, but this book uses appropriate access modifiers with other methods.

## An Introduction to the Optional `static` Modifier

You can declare a method to be **static** or **nonstatic**. If you use the keyword modifier `static`, you indicate that a method can be called without referring to an object. Instead, you refer only to the class. For example, if `MethodS()` is static and `MethodN()` is nonstatic, the following statements describe typical method calls:

```
someClass.MethodS();  
someObject.MethodN();
```



You won't create objects until Chapter 9, so this chapter will use only nonstatic methods.

A nonstatic method might be called with or without its class name. For example, if you have a class named `PayrollApplication` that contains a static method named `DeductUnionDues()`, you can call the method in two ways:

- If you call the method from a method in a different class, you use the class name, a dot, and the method name, as in the following:

```
PayrollApplication.DeductUnionDues();
```

- If you call the method from another method in the same class, you *can* use the class name as a prefix as shown above, but this approach is neither required nor conventional. Within a method in the `PayrollApplication` class, you can simply write the following abbreviated method call:

```
DeductUnionDues();
```

If you do not indicate that a method is static, it is nonstatic by default and can be used only in conjunction with an object. When you begin to create your own classes in the chapter “Using Classes and Objects,” you will write many nonstatic methods, and your understanding of the terms *static* and *nonstatic* will become clearer. As you work through this chapter, you will learn why each method is created to be static or nonstatic. Do not worry if you do not completely understand this chapter’s references to nonstatic methods that require an object—the concept is explored in Chapter 9.

## An Introduction to Return Types

Every method has a **return type**, indicating what kind of value the method will return to any other method that calls it. If a method does not return a value, its return type is `void`. A method’s return type is known more succinctly as a **method’s type**. Later in this chapter, you will create methods with return types such as `int` or `double` that return values of the corresponding type; such methods can be called *value-returning methods*. For now, the methods discussed are `void` methods that do not return a value.



When a method’s return type is `void`, most C# programmers do not end the method with a `return` statement. However, you can end a `void` method with the following statement that indicates nothing is returned:

```
return;
```

## Understanding the Method Identifier

Every method has a name that must be a legal C# identifier; that is, it must not contain spaces and must begin with a letter of the alphabet or an underscore. By convention, many programmers start method names with a verb because methods cause actions. Examples of conventional method names include `DeductTax()` and `DisplayNetPay()`.

Every method name is followed by a set of parentheses. Sometimes these parentheses contain parameters, but in the simplest methods, the parentheses are empty. A **parameter to a method** is a variable that holds data passed to a method when it is called. The terms *argument* and *parameter* are closely related. An argument is data in a method call, and a parameter is in the method header—it receives an argument's value when the method executes.

The parentheses that follow a method name in its header can hold one parameter or multiple parameters separated with commas. The contents within the parentheses are known as the **parameter list**.

## Placing a Method in a Class

In summary, the first methods you write in console applications will be `private`, `static`, and `void` and will have empty parameter lists. That means they won't be called from other classes, they won't require an object reference, they will not return any value to their calling method, and they will not accept any data from the outside. Therefore, you can write the `ShowWelcomeMessage()` method as it is shown in Figure 7-2. According to its declaration, the method is `private` and `static`. It returns nothing, so the return type is `void`. Its identifier is `ShowWelcomeMessage`, and it receives nothing, so its parameter list is empty. The method body, consisting of three `WriteLine()` statements, appears within curly braces that follow the header.

By convention, programmers indent the statements in a method body, which makes the method header and its braces stand out. When you write a method using the Visual Studio editor, the method statements are indented for you automatically. You can place as many statements as you want within a method body.

You also can place a method within a class in which other methods will use it, but you cannot place a method within any other method. Figure 7-3 shows the two locations where you can place the `ShowWelcomeMessage()` method within the class named `HelloClass`—before the `Main()` method header or after the `Main()` method's closing brace but before the closing curly brace for the class.

```
using static System.Console;
class HelloClass
{
    // The ShowWelcomeMessage() method could go here
    static void Main()
    {
        WriteLine("Hello");
    }
    // Alternatively, the ShowWelcomeMessage() method could go here
    // But it cannot go in both places; it can appear only once
}
```

**Figure 7-3** Placement of a method



To make the `Main()` method call the `ShowWelcomeMessage()` method, you simply use the `ShowWelcomeMessage()` method's name as a statement within the body of the `Main()` method. Figure 7-4 shows the complete program with the method call shaded, and Figure 7-5 shows the output.

```
using static System.Console;
class HelloClass
{
    static void Main()
    {
        ShowWelcomeMessage();
        WriteLine("Hello");
    }
    private static void ShowWelcomeMessage()
    {
        WriteLine("Welcome.");
        WriteLine("Have fun!");
        WriteLine("Enjoy the program!");
    }
}
```

**Figure 7-4** The `HelloClass` program with `Main()` method calling the `ShowWelcomeMessage()` method

```
C:\C#\Chapter.07>HelloClass
Welcome.
Have fun!
Enjoy the program!
Hello
C:\C#\Chapter.07>
```

**Figure 7-5** Output of the `HelloClass` program

The `ShowWelcomeMessage()` method in the `HelloClass` class is `static`, and therefore it is called from the `Main()` method without an object reference (that is, without an object name and a dot before the method name). It also resides in the same class as the `Main()` method, so it can be called without using its class name.

When the `Main()` method executes, it calls the `ShowWelcomeMessage()` method, so the three lines that make up the welcome message appear first in the output in Figure 7-5. Then, after the method is done, the `Main()` method displays *Hello*.

Each of two different classes can have its own method named `ShowWelcomeMessage()`. Such a method in the second class would be entirely distinct from the identically named method in the first class. The complete name of this method is `HelloClass.ShowWelcomeMessage()`, but you do not need to use the complete name when calling the method within the same class.

If another class named `SomeOtherClass` had a *public* static method with the same name, you could call the method from `HelloClass` using the following statement:

```
SomeOtherClass.ShowWelcomeMessage();
```

## Declaring Variables and Constants in a Method

You can write any statements you need within a method, including variable and constant declarations. When a variable or constant is declared within a method, it is known only from that point to the end of the method; programmers say the variable or constant is in scope until the end of its method. A program element's **scope** is the segment of code in which it can be used. Programmers also say that the variable is **local** to the method, meaning it can be used anywhere in the method after the point of declaration.

A locally declared variable is not known to other methods or usable in them, and if another method contains a variable with the same name, the two variables are completely distinct. For example, Figure 7-6 shows a program containing two methods—`Main()` and `MethodWithItsOwnA()`—that each declare a variable named `a`. The variable in each method is completely distinct from the other and holds its own value, as shown in Figure 7-7. If you declared a variable named `b` in the `Main()` method and then tried to use `b` in `MethodWithItsOwnA()`, you would generate a compiler error. If you declared a variable named `c` in `MethodWithItsOwnA()` and tried to use it in `Main()`, you would generate another compiler error.

```
using static System.Console;
class LocalVariableDemo
{
    static void Main()
    {
        int a = 12;
        WriteLine("In Main() a is {0}", a);
        MethodWithItsOwnA();
        WriteLine("In Main() a is {0}", a);
    }
    private static void MethodWithItsOwnA()
    {
        int a = 354;
        WriteLine("In method a is {0}", a);
    }
}
```

**Figure 7-6** The `LocalVariableDemo` program

```
C:\C#\Chapter.07>LocalVariableDemo
In Main() a is 12
In method a is 354
In Main() a is 12
C:\C#\Chapter.07>_
```

**Figure 7-7** Execution of the `LocalVariableDemo` program



Watch the video *Using Methods*.

## TWO TRUTHS & A LIE

### Writing Methods with No Parameters and No Return Value

1. A method header must contain declared accessibility.
2. A method header must contain a return type.
3. A method header must contain an identifier.

The false statement is #1. Declaring accessibility in a method header is optional. If you do not use an access modifier, the method is private by default.



### You Do It

#### Calling a Method

In this section, you write a program in which a `Main()` method calls another method that displays a company's logo.

1. Open a new program named **DemoLogo**, and enter the statement that allows the program to use `System.Console` methods without qualifying them. Then type the class header for the `DemoLogo` class and the class-opening curly brace.

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

2. Type the `Main()` method for the `DemoLogo` class. This method displays a line, then calls the `DisplayCompanyLogo()` method.

```
static void Main()
{
    Write("Our company is ");
    DisplayCompanyLogo();
}
```

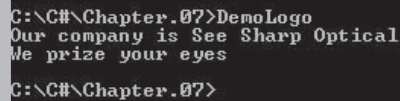
(continues)

(continued)

3. Add a method that displays a two-line logo for a company.

```
private static void DisplayCompanyLogo()
{
    WriteLine("See Sharp Optical");
    WriteLine("We prize your eyes");
}
```

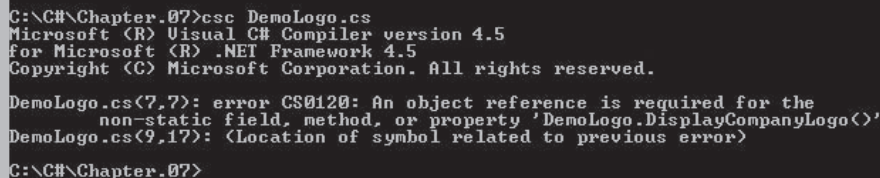
4. Add the closing curly brace for the class ( `}` ), and then save the file.
5. Compile and execute the program. The output should look like Figure 7–8.



```
C:\C#\Chapter.07>DemoLogo
Our company is See Sharp Optical
We prize your eyes
C:\C#\Chapter.07>
```

**Figure 7-8** Output of the DemoLogo program

6. Remove the keyword `static` from the `DisplayCompanyLogo()` method, and then save and compile the program. An error message appears, as shown in Figure 7-9. The message indicates that an object reference is required for the nonstatic `DisplayCompanyLogo()` method. The error occurs because the `Main()` method is `static` and cannot call a nonstatic method without an object reference (that is, without using an object name and a dot before the method call). Remember that you will learn to create objects in Chapter 9, and then you can create the types of methods that do not use the keyword `static`. For now, retype the keyword `static` in the `DisplayCompanyLogo()` method header, and compile and execute the program again.



```
C:\C#\Chapter.07>csc DemoLogo.cs
Microsoft (R) Visual C# Compiler version 4.5
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

DemoLogo.cs(7,7): error CS0120: An object reference is required for the
non-static field, method, or property 'DemoLogo.DisplayCompanyLogo()'
DemoLogo.cs(9,17): <Location of symbol related to previous error>

C:\C#\Chapter.07>
```

**Figure 7-9** Error message when calling a nonstatic method from a static method

7. Remove the keyword `private` from the header of the `DisplayCompanyLogo()` method. Save and compile the program, and then execute it. The execution is successful because the `private` keyword is optional. Replace the `private` keyword, and save the program.

## Writing Methods That Require a Single Argument

Some methods require additional information. If a method could not receive arguments, then you would have to write an infinite number of methods to cover every possible situation. For example, when you make a dental appointment, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the method, and no matter what date and time you supply, the method is carried out correctly. If you design a method to compute an employee's paycheck, it makes sense that you can write a method named `ComputePaycheck()` and supply it with an hourly pay rate rather than having to develop methods with names like `ComputePaycheckAtEightDollarsAnHour()`, `ComputePaycheckAtNineDollarsAnHour()`, and so on.

You already have used methods to which you supplied a wide variety of parameters. At any call, the `WriteLine()` method can receive any one of an infinite number of strings as a parameter—"Hello", "Goodbye", and so on. No matter what message you send to the `WriteLine()` method, the message is displayed correctly.

When you write the declaration for a method that accepts a parameter, you need to include the following items within the method declaration parentheses:

- The data type of the parameter
- A local identifier (name) for the parameter

For example, consider a method named `DisplaySalesTax()`, which computes and displays a tax as 7 percent of a selling price. The method header for a usable `DisplaySalesTax()` method could be the following:

```
static void DisplaySalesTax(double saleAmount)
```

You can think of the parentheses in a method declaration as a funnel into the method—data parameters listed there are “dropping in” to the method.

The parameter `double saleAmount` within the parentheses indicates that the `DisplaySalesTax()` method will receive a value of type `double`. Within the method, the value will be known as `saleAmount`. Figure 7-10 shows a complete method.

```
private static void DisplaySalesTax(double saleAmount)
{
    double tax;
    const double RATE = 0.07;
    tax = saleAmount * RATE;
    WriteLine("The tax on {0} is {1}",
        saleAmount, tax.ToString("C"));
}
```

**Figure 7-10** The `DisplaySalesTax()` method



Within the `DisplaySalesTax()` method, you must use the format string and `ToString()` method if you want figures to display to exactly two decimal positions. You learned how to display values to a fixed number of decimal places in Chapter 2; recall that using the fixed format with no number defaults to two decimal places.

You create the `DisplaySalesTax()` method as a `void` method (one that has a `void` return type) because you do not need it to return any value to any method that uses it—its only function is to receive the `saleAmount` value, multiply it by 0.07, and then display the result. You create it as a `static` method because you do not want to create an object with which to use it; you want the `static Main()` method to be able to call it directly.

Within a program, you can call the `DisplaySalesTax()` method by using the method's name and, within parentheses, an argument that is either a constant value or a variable. Thus, both of the following calls to the `DisplaySalesTax()` method invoke it correctly:

```
double myPurchase = 12.99;
DisplaySalesTax(12.99);
DisplaySalesTax(myPurchase);
```

You can call the `DisplaySalesTax()` method any number of times, with a different constant or variable argument each time. The value of each of these arguments becomes known as `saleAmount` within the method. Interestingly, if the argument in the method call is a variable, it might possess the same identifier as `saleAmount` or a different one, such as `myPurchase`. The identifier `saleAmount` is simply the name the value “goes by” while being used within the method, no matter what name it goes by in the calling program. That is, the variable `saleAmount` is a local variable to the `DisplaySalesTax()` method, as are variables and constants declared within a method. The variable `saleAmount` is also an example of a **formal parameter**, a parameter within a method header that accepts a value. In contrast, arguments within a method *call* often are referred to as **actual parameters**. For example, in the method calling statement `DisplaySalesTax(myPurchase);`, `myPurchase` is an actual parameter.



The variable `saleAmount` is a formal parameter, and it also is an example of a *value parameter*, or a *parameter* that receives a copy of the value passed to it. You will learn more about value parameters and other types of parameters in the next chapter.

The `DisplaySalesTax()` method employs implementation hiding. That is, if a programmer changes the way in which the tax value is calculated—for example, by coding one of the following—programs that use the `DisplaySalesTax()` method will not be affected and will not need to be modified:

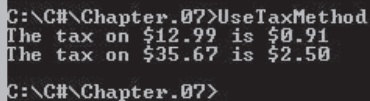
```
tax = saleAmount * 7 / 100;
tax = 0.07 * saleAmount;
tax = RATE * saleAmount;
```

No matter how the tax is calculated, a calling program passes a value into the `DisplaySalesTax()` method, and a calculated result appears on the screen.

Figure 7-11 shows a complete program called `UseTaxMethod`. It uses the `DisplaySalesTax()` method twice, first with a variable argument, and then with a constant argument. The program's output appears in Figure 7-12.

```
using static System.Console;
class UseTaxMethod
{
    static void Main()
    {
        double myPurchase = 12.99;
        DisplaySalesTax(myPurchase);
        DisplaySalesTax(35.67);
    }
    private static void DisplaySalesTax(double saleAmount)
    {
        double tax;
        const double RATE = 0.07;
        tax = saleAmount * RATE;
        WriteLine("The tax on {0} is {1}",
            saleAmount.ToString("C"), tax.ToString("C"));
    }
}
```

**Figure 7-11** Complete program using the `DisplaySalesTax()` method two times



```
C:\C#\Chapter.07>UseTaxMethod
The tax on $12.99 is $0.91
The tax on $35.67 is $2.50
C:\C#\Chapter.07>
```

**Figure 7-12** Output of the `UseTaxMethod` program

An argument type in a method call can match the method's parameter type exactly, but it can use a different data type if the argument can be converted automatically to the parameter type. Recall from Chapter 2 that C# supports the following automatic conversions:

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

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

As an example, a method with the header `private static void DisplaySalesTax(double saleAmount)` can work with the following method call that uses an integer argument because integers are promoted automatically to doubles:

```
DisplaySalesTax(100);
```



Now that you have seen how to write methods that accept an argument, you might guess that when you write `writeLine("Hello");`, the header for the called method is similar to `public static void writeLine(string s)`. You might not know the parameter name the creators of C# have chosen, but you do know the method's return type, name, and parameter type. (If you use the IntelliSense feature of Visual Studio, you can discover the parameter name. See Appendix C for more details.)



Watch the video *Arguments and Parameters*.

## TWO TRUTHS & A LIE

### Writing Methods That Require a Single Argument

1. When you write the declaration for a method that accepts a parameter, you need to include the parameter's data type within the method header.
2. When you write the declaration for a method that accepts a parameter, you need to include the identifier of the argument that will be sent to the method within the method header.
3. When you write the declaration for a method that accepts a parameter, you need to include a local identifier for the parameter within the method header.

The false statement is #2. When you write the definition for a method, you include the data type and a local parameter name within the parentheses of the method header, but you do not include the name of any argument that will be sent from a calling method. After all, the method might be invoked any number of times with any number of different arguments.

## Writing Methods That Require Multiple Arguments

You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas. For example, rather than creating a `DisplaySalesTax()` method that multiplies an amount by 0.07, you might prefer to create a more flexible method to which you can pass two values—the value on which the tax is calculated and the tax percentage by which it should be multiplied. Figure 7-13 shows a method that uses two parameters.



```
private static void DisplaySalesTax(double saleAmount, double taxRate)
{
    double tax;
    tax = saleAmount * taxRate;
    WriteLine("The tax on {0} at {1} is {2}",
        saleAmount.ToString("C"), taxRate.ToString("P"),
        tax.ToString("C"));
}
```

**Figure 7-13** The `DisplaySalesTax()` method that accepts two parameters

In Figure 7-13, two parameters (`saleAmount` and `taxRate`) appear within the parentheses in the method header. A comma separates the parameters, and each parameter requires its own named type (in this case, both parameters are of type `double`) and an identifier. A declaration for a method that receives two or more arguments must list the type for each parameter separately, even if the parameters have the *same* type.

When you pass values to the method in a statement such as `DisplaySalesTax(myPurchase, localRate);`, the first value passed will be referenced as `saleAmount` within the method, and the second value passed will be referenced as `taxRate`. Therefore, it is very important that arguments be passed to a method in the correct order. The following call results in output stating that *The tax on \$200.00 at 10.00% is \$20.00*:

```
DisplaySalesTax(200.00, 0.10);
```

However, the following call results in output stating that *The tax on \$0.10 at 10.00% is \$20.00*, which is clearly incorrect.

```
DisplaySalesTax(0.10, 200.00);
```

You can write a method to take any number of parameters in any order. When you call the method, however, the arguments you send to it must match (in both number and type) the parameters listed in the method declaration, with the following exceptions:

- The type of an argument does not need to match the parameter list exactly if the argument can be promoted to the parameter type. For example, an `int` argument can be passed to a `double` parameter.
- The number of arguments does not need to match the number in the parameter list when you use *default arguments*. You will learn about default arguments in Chapter 8.

Thus, a method to compute and display an automobile salesperson's commission might require arguments such as a string for the salesperson's name, an integer value of a sold car, a `double` percentage commission rate, and a character code for the vehicle type. The correct method will execute only when all arguments of the correct types are sent in the correct order.

## TWO TRUTHS & A LIE

### Writing Methods That Require Multiple Arguments

1. The following is a usable C# method header:

```
private static void MyMethod(double amt, sum)
```

2. The following is a usable C# method header:

```
private static void MyMethod2(int x, double y)
```

3. The following is a usable C# method header:

```
static void MyMethod3(int id, string name, double rate)
```

The false statement is #1. In a method header, each parameter must have a data type, even if the data types for all the parameters are the same. The header in #3 does not contain an accessibility indicator, but that is optional.

## Writing a Method That Returns a Value

A method can return, at most, one value to a method that calls it. The return type for a method can be any type used in the C# programming language, which includes the basic built-in types `int`, `double`, `char`, and so on. You can also use a class type as the return type for a method, including any type built into the C# language. For example, you have used a return value from the `ReadLine()` method when writing a statement such as `inputString = ReadLine();`. Because the `ReadLine()` method call can be assigned to a `string`, its return type is `string`. You will learn to create your own types in Chapter 9, and these might also be returned from a method. Of course, a method also can return nothing, in which case the return type is `void`.

For example, suppose that you want to create a method to accept the hours an employee worked and the hourly pay rate, and to return a calculated gross pay value. The header for this method could be:

```
private static double CalcPay(double hours, double rate)
```

Figure 7-14 shows this method.

```
private static double CalcPay(double hours, double rate)
{
    double gross;
    gross = hours * rate;
    return gross;
}
```

**Figure 7-14** The `CalcPay()` method

Notice the **return** statement, which is the last statement within the `CalcPay()` method. A **return statement** causes a value to be sent back to the calling method; in the `CalcPay()` method, the value stored in `gross` is sent back to any method that calls the `CalcPay()` method. Also notice the type `double` that precedes the method name in the method header. The data type used in a method's **return** statement must be the same as the return type declared in the method's header, or the program will not compile.

If a method returns a value and you call the method, you typically will want to use the returned value, although you are not required to use it. For example, when you invoke the `CalcPay()` method, you might want to assign the value to a `double` variable named `grossPay`, as in the following statement:

```
grossPay = CalcPay(myHours, myRate);
```

The `CalcPay()` method returns a `double`, so it is appropriate to assign the returned value to a `double` variable. Figure 7-15 shows a program that uses the `CalcPay()` method in the shaded statement, and Figure 7-16 shows the output.

```
using static System.Console;
class UseCalcPay
{
    static void Main()
    {
        double myHours = 37.5;
        double myRate = 12.75;
        double grossPay;
        grossPay = CalcPay(myHours, myRate);
        WriteLine("I worked {0} hours at {1} per hour",
            myHours, myRate);
        WriteLine("My gross pay is {0}", grossPay.ToString("C"));
    }
    private static double CalcPay(double hours, double rate)
    {
        double gross;
        gross = hours * rate;
        return gross;
    }
}
```

**Figure 7-15** Program using the `CalcPay()` method

```
C:\C#\Chapter.07>UseCalcPay
I worked 37.5 hours at 12.75 per hour
My gross pay is $478.13

C:\C#\Chapter.07>
```

**Figure 7-16** Output of the `UseCalcPay` program

Instead of storing a method's returned value in a variable, you can use it directly, as in statements that produce output or perform arithmetic such as the following:

```
WriteLine("My gross pay is {0}",  
    CalcPay(myHours, myRate).ToString("C"));  
double tax = CalcPay(myHours, myRate) * TAX_RATE;
```

In the first statement, the call to the `CalcPay()` method is made within the `WriteLine()` method call. In the second, `CalcPay()`'s returned value is used in an arithmetic statement. Because `CalcPay()` returns a `double`, you can use the method call `CalcPay()` in the same way you would use any `double` value. The method call `CalcPay()` has a `double` data type in the same way a `double` variable does.

As an additional example, suppose that you have a method named `GetPrice()` that accepts an item number and returns its price. The header is as follows:

```
private static double GetPrice(int itemNumber)
```

Further suppose that you want to ask the user to enter an item number from the keyboard so you can pass it to the `GetPrice()` method. You can get the value from the user, store it in a string, convert the string to an integer, pass the integer to the `GetPrice()` method, and store the returned value in a variable named `price` in four or five separate statements. Or you can write the following:

```
price = GetPrice(Convert.ToInt32(ReadLine()));
```

This statement contains a method call to `ReadLine()` within a method call to `Convert.ToInt32()`, within a method call to `GetPrice()`. When method calls are placed inside other method calls, the calls are **nested method calls**. When you write a statement with three nested method calls like the previous statement, the innermost method executes first. Its return value is then used as an argument to the intermediate method, and its return value is used as an argument to the outer method. There is no limit to how “deep” you can go with nested method calls.



The system keeps track of where to return after a method call in an area of memory called the *stack*. Another area of memory called the *heap* is where memory can be allocated while a program is executing.

## Writing a Method That Returns a Boolean Value

When a method returns a value that is type `bool`, the method call can be used anywhere you can use a Boolean expression. For example, suppose you have written a method named `isPreferredCustomer()` that returns a Boolean value indicating whether a customer is a preferred customer who qualifies for a discount. Then you can write an `if` statement such as the following:

```
if(isPreferredCustomer())  
    price = price - DISCOUNT;
```

In Chapter 4 you learned about side effects and how they affect compound Boolean expressions. When you use Boolean methods, you must be especially careful not to cause unintended side effects. For example, consider the following `if` statement, in which the intention is to set a delivery fee to 0 if both the `isPreferredCustomer()` and `isLocalCustomer()` methods return true:

```
if(isPreferredCustomer() && isLocalCustomer())
    deliveryFee = 0;
```

If the `isLocalCustomer()` method should perform some desired task—for example, displaying a message about the customer’s status or applying a local customer discount to the price—then you might not achieve the desired results. Because of short-circuit evaluation, if the `isPreferredCustomer()` method returns `false`, the `isLocalCustomer()` method never executes. If that is your intention, then using methods in this way is fine, but always consider any unintended side effects.

## Analyzing a Built-In Method

C# provides you with many prewritten methods such as `WriteLine()` and `ReadLine()`. In Chapter 2, you learned about C#’s arithmetic operators, such as `+` and `*`, and you learned that C# provides no exponential operator. Instead, to raise a number to a power, you can use the built-in `Pow()` method. For example, to raise 2 to the third power ( $2 * 2 * 2$ ) and store the answer in the variable `result`, you can write a program that includes the following statement:

```
double result = Math.Pow(2.0, 3.0);
```

From this statement, you know the following about the `Pow()` method:

- It is in the `Math` class because the class name and a dot precede the method call.
- It is `public` because you can write a program that uses it.
- It is `static` because it is used with its class name and a dot, without any object.
- It accepts two `double` parameters.
- It returns a `double` or a type that can be promoted automatically to a `double` (such as an `int`) because its answer is stored in a `double`.

Although you know many facts about the `Pow()` method, you do not know how its instructions are carried out internally. In good object-oriented style, its implementation is hidden.

## TWO TRUTHS & A LIE

### Writing a Method That Returns a Value

1. A method can return, at most, one value to a method that calls it.
2. The data type used in a method's return statement must be the same as the return type declared in the method's header.
3. If a method returns a value and you call the method, you must store the value in a variable that has the same data type as the method's parameter.

The false statement is #3. If a method returns a value and you call the method, you typically will want to use the returned value, but you are not required to use it. Furthermore, if you do assign the returned value to a storage variable, that variable does not have to match any parameter's value. Instead, the storage variable needs to be capable of accepting the method's returned type.



### You Do It

#### Writing a Method That Receives Parameters and Returns a Value

Next, you write a method named `CalcPhoneCallPrice()` that both receives parameters and returns a value. The purpose of the method is to accept the length of a phone call in minutes and the rate charged per minute and to then calculate the price of a call, assuming each call includes a 25-cent connection charge in addition to the per-minute charge. After writing the `CalcPhoneCallPrice()` method, you write a `Main()` method that calls the `CalcPhoneCallPrice()` method using four different sets of data as arguments.

1. Start a program named **PhoneCall** by typing a `using` statement, class header, and opening brace:

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

(continues)

*(continued)*

2. Type the following `CalcPhoneCallPrice()` method. The method is declared as `static` because it will be called by a `static Main()` method without creating an object. The method receives an `integer` and a `double` as parameters. The fee for a call is calculated as 0.25 plus the minutes times the rate per minute. The method returns the phone call fee to the calling method.

```
private static double CalcPhoneCallPrice(int minutes,
    double rate)
{
    const double BASE_FEE = 0.25;
    double callFee;
    callFee = BASE_FEE + minutes * rate;
    return callFee;
}
```

3. Add the `Main()` method header for the `PhoneCall` class. Begin the method by declaring two arrays; one contains two call lengths, and the other contains two rates. You will use all the possible combinations of call lengths and rates to test the `CalcPhoneCallPrice()` method. Also, declare a `double` named `priceOfCall` that will hold the result of a calculated call price.

```
static void Main()
{
    int[] callLengths = {2, 5};
    double[] rates = {0.03, 0.12};
    double priceOfCall;
```

4. Add a statement that displays column headings under which you can list combinations of call lengths, rates, and prices. The three column headings are right-aligned, each in a field 10 characters wide.

```
WriteLine("{0, 10}{1, 10}{2, 10}",
    "Minutes", "Rate", "Price");
```

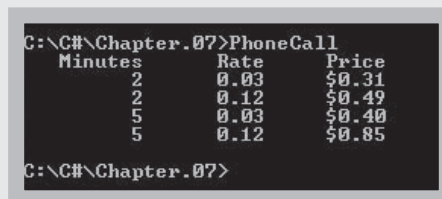
*(continues)*

*(continued)*

5. Add a pair of nested loops that pass each `callLength` and each `rate` to the `CalcPhoneCallPrice()` method in turn. As each pair is passed, the result is stored in the `priceOfCall` variable, and the details are displayed. Using the nested loops allows you to pass each combination of call time and rate so that multiple possibilities for the values can be tested conveniently.

```
for(int x = 0; x < callLengths.Length; ++x)
    for(int y = 0; y < rates.Length; ++y)
    {
        priceOfCall = CalcPhoneCallPrice(callLengths[x],
            rates[y]);
        WriteLine("{0, 10}{1, 10}{2, 10}",
            callLengths[x], rates[y], priceOfCall.ToString("C"));
    }
```

6. Add a closing curly brace for the `Main()` method and another for the `PhoneCall` class.
7. Save the file, and then compile and run the program. The output looks like Figure 7-17. It shows how a single method can produce a variety of results when you use different values for the arguments.



```
C:\C#\Chapter.07>PhoneCall
Minutes    Rate      Price
2          0.03     $0.31
2          0.12     $0.49
5          0.03     $0.40
5          0.12     $0.85

C:\C#\Chapter.07>
```

**Figure 7-17** Output of the `PhoneCall` program



## Passing Array Values to a Method

Passing an array element to a method and passing an array to a method require different approaches.

### Passing a Single Array Element to a Method

In the chapter “Using Arrays,” you learned that you can declare an array to create a list of elements, and that you can use any individual array element in the same manner as you would use any single variable of the same type. That is, suppose you declare an integer array as follows:

```
int[] someNums = new int[12];
```

You can subsequently output `someNums[0]` or add 1 to `someNums[1]`, just as you would for any integer. Similarly, you can pass a single array element to a method in exactly the same manner as you would pass a variable.

Consider the program shown in Figure 7-18. This program creates and uses an array of four integers. Figure 7-19 shows the program’s execution.

```
using static System.Console;
class PassArrayElement
{
    static void Main()
    {
        int[] someNums = {10, 12, 22, 35};
        int x;
        Write("\nAt beginning of Main() method...");
        for(x = 0; x < someNums.Length; ++x)
            Write("{0,6}", someNums[x]);
        WriteLine();
        for(x = 0; x < someNums.Length; ++x)
            MethodGetsOneInt(someNums[x]);
        Write("At end of Main() method.....");
        for(x = 0; x < someNums.Length; ++x)
            Write("{0,6}", someNums[x]);
    }
    private static void MethodGetsOneInt(int oneVal)
    {
        Write("In MethodGetsOneInt() {0}", oneVal);
        oneVal = 999;
        WriteLine("    After change {0}", oneVal);
    }
}
```

**Figure 7-18** The `PassArrayElement` program

```
C:\C#\Chapter.07>PassArrayElement
At beginning of Main() method...   10   12   22   35
In MethodGetsOneInt() 10   After change 999
In MethodGetsOneInt() 12   After change 999
In MethodGetsOneInt() 22   After change 999
In MethodGetsOneInt() 35   After change 999
At end of Main() method.....   10   12   22   35
C:\C#\Chapter.07>
```

**Figure 7-19** Output of the PassArrayElement program

As you can see in Figure 7-19, the program displays the four original values, then passes each to the `MethodGetsOneInt()` method, where it is displayed and then changed to 999. After the method executes four times, the `Main()` method displays the four values again, showing that they are unchanged by the assignments within `MethodGetsOneInt()`. The `oneVal` variable is local to the `MethodGetsOneInt()` method; therefore, any changes to variables passed into the method are not permanent and are not reflected in the array declared in the `Main()` program. Each time the `MethodGetsOneInt()` method executes, its `oneVal` parameter holds only a copy of the array element passed into the method, and the `oneVal` variable exists only while the `MethodGetsOneInt()` method is executing.

## Passing an Array to a Method

Instead of passing a single array element to a method, you can pass an entire array. You indicate that a method parameter must be an array by placing square brackets after the data type in the method's parameter list. When you pass an array to a method, changes you make to array elements within the method are reflected in the original array that was sent to the method. Arrays, like all objects but unlike built-in types such as `double` and `int`, are **passed by reference**. That is, when a simple built-in type is passed to a method, the method receives a copy of the value at a new memory address, but when an array is passed to a method, the method receives the actual memory address of the array and has access to the actual values in the array elements. You already have seen that methods can alter arrays passed to them. When you use the `Sort()` and `Reverse()` methods, the methods change the array parameter contents.

The program shown in Figure 7-20 creates an array of four integers. After the integers are displayed, the entire array is passed to a method named `MethodGetsArray()` in the shaded statement. Within the method header, the parameter is declared as an array by using square brackets after the parameter data type. Within the method, the numbers are output, which shows that they retain their values from `Main()` upon entering the method, but then the value 888 is assigned to each number. Even though `MethodGetsArray()` is a `void` method (meaning that nothing is returned to the `Main()` method), when the program displays the array for the second time within the `Main()` method, all of the values have been changed to 888, as you can see in Figure 7-21. Because arrays are passed by reference, the `MethodGetsArray()` method “knows” the address of the array declared in `Main()` and makes its changes directly to the original array that was declared in the `Main()` method.

```

using static System.Console;
class PassEntireArray
{
    static void Main()
    {
        int[] someNums = {10, 12, 22, 35};
        int x;
        Write("\nAt beginning of Main() method...");
        for(x = 0; x < someNums.Length; ++x)
            Write("{0, 6}", someNums[x]);
        WriteLine();
        MethodGetsArray(someNums);
        Write("At end of Main() method.....");
        for(x = 0; x < someNums.Length; ++x)
            Write("{0, 6}", someNums[x]);
    }
    private static void MethodGetsArray(int[] vals)
    {
        int x;
        Write("In MethodGetsArray() ");
        for(x = 0; x < vals.Length; ++x)
            Write(" {0}", vals[x]);
        WriteLine();
        for(x = 0; x < vals.Length; ++x)
            vals[x] = 888;
        Write("After change");
        for(x = 0; x < vals.Length; ++x)
            Write(" {0}", vals[x]);
        WriteLine();
    }
}

```

**Figure 7-20** The PassEntireArray program

```

C:\C#\Chapter.07>PassEntireArray
At beginning of Main() method...    10    12    22    35
In MethodGetsArray()  10 12 22 35
After change 888 888 888 888
At end of Main() method.....    888    888    888    888
C:\C#\Chapter.07>_

```

**Figure 7-21** Output of the PassEntireArray program

Notice that you do not insert a number within the square brackets of the array definition in the method parameter list. Inserting a number causes a compiler error. It makes sense that the brackets are empty because a method that receives an array gets its starting address, not a number of elements.



You can create and pass an unnamed array to a method in a single step. For example, you can write the following:

```
MethodThatAcceptsArray(new int[] {45, 67, 89});
```

You can pass a multidimensional array to a method by indicating the appropriate number of dimensions after the data type in the method header. For example, the following method headers accept two-dimensional arrays of `ints` and `doubles`, respectively:

```
private static void displayScores(int[ , ] scoresArray)
private static boolean areAllPricesHigh(double[ , ] prices)
```

With jagged arrays, you can insert the appropriate number of square brackets after the data type in the method header. For example, the following method headers accept jagged arrays of `ints` and `doubles`, respectively:

```
private static void displayIDs(int[][] idArray)
private static double computeTotal(double[][] prices)
```

With methods that accept multidimensional arrays as parameters (whether jagged or not), notice that the brackets that define the array in each method header are empty. Like any other array passed to a method, inserting numbers into the brackets for a jagged array is not necessary because each passed array name is a starting memory address. The way you manipulate subscripts within the method determines how rows and columns are accessed.



The size of each dimension of a multidimensional array can be accessed using the `GetLength()` method. For example, `scoresArray.GetLength(0)` returns the value of the first dimension of `scoresArray`.



Watch the video *Passing Arrays to Methods*.

## TWO TRUTHS & A LIE

### Passing Array Values to a Method

1. You indicate that a method parameter can be an array element by placing a data type and identifier in the method's parameter list.
2. You indicate that a method parameter must be an array by placing parentheses after the data type in the method's parameter list.
3. Arrays are passed by reference; that is, the method receives the actual memory address of the array and has access to the actual values in the array elements.

The false statement is #2. You indicate that a method parameter must be an array by placing square brackets after the data type in the method's parameter list.

## Alternate Ways to Write a Main() Method Header

Throughout this book, you have written `Main()` methods with the following header:

```
static void Main()
```

Using the return type `void` and listing nothing between the parentheses that follow `Main` is just one way to write a `Main()` method header in a program. However, it is the first way listed in the C# documentation, and it is the convention used in this book. This section describes alternatives because you might see different `Main()` method headers in other books or in programs written by others.

### Writing a Main() Method with a Parameter List

An alternate way to write a `Main()` method header is as follows:

```
static void Main(string[] args)
```

The phrase `string[] args` is a parameter to the `Main()` method. The variable `args` represents an array of strings that you can pass to `Main()`. Although you can use any identifier, `args` is conventional. In particular, Java programmers might prefer the C# version of `Main()` that includes the `string[] args` parameter because their convention is to write `main` methods with the same parameter.

Use this format for the `Main()` method header if you need to access command-line arguments passed in to your application. For example, the program in Figure 7-22 displays an `args` array that is a parameter to its `Main()` method. Figure 7-23 shows how a program might be executed from the command line using arguments to `Main()`.

```
using static System.Console;
class DisplayArgs
{
    static void Main(string[] args)
    {
        for(int x = 0; x < args.Length; ++x)
            WriteLine("Argument {0} is {1}", x, args[x]);
    }
}
```

**Figure 7-22** A `Main()` method with a `string[] args` parameter

```
C:\C#\Chapter.07>DisplayArgs Alpha Beta Gamma Delta
Argument 0 is Alpha
Argument 1 is Beta
Argument 2 is Gamma
Argument 3 is Delta
C:\C#\Chapter.07>_
```

**Figure 7-23** Executing the `DisplayArgs` program with arguments

Even if you do not need access to command-line arguments, you can still use the version of the `Main()` method header that references them. You should use this version if your instructor or supervisor indicates you should follow this convention.

## Writing a `Main()` Method with an Integer Return Type

Some programmers prefer to write `Main()` method headers that have a return type of `int` instead of `void`. If you use this form, the last statement in the `Main()` method must be a `return` statement that returns an integer. By convention, a return value of 0 means that an application ended without error. The value might be used by your operating system or another program that uses your program. In particular, C++ programmers might prefer the version of `Main()` that returns an `int` because conventionally they write their main methods with an `int` return type.

Even if you do not need to use a return value from a `Main()` method, you can still use the version of the `Main()` method header that includes a return value. You should use this version if your instructor or supervisor indicates you should follow this convention.

## Writing a `Main()` Method with `public` Access

Some programmers prefer to write `Main()` method headers with `public` access rather than `private`, which is the default access when no specifier is listed. In particular, Java programmers might prefer using `public` because the style is conventional for them. If a class's `Main()` method is `public`, it can be called from another class by using its fully qualified name. Because you rarely want to do this, C# programmers often omit the `Main()` method's access specifier, making it `private`.

### TWO TRUTHS & A LIE

#### Alternate Ways to Write a `Main()` Method Header

1. In C#, a `Main()` method header can be written `public static void Main()`.
2. In C#, a `Main()` method header can be written `static void Main(string[] args)`.
3. In C#, a `Main()` method header can be written `static int main(string args)`.

The false statement is #3. In C#, a `Main()` method header can be written as shown in either of the first two examples. Statement #3 is wrong because `Main()` must be capitalized, and `string` must be followed by a pair of square brackets.

## Issues Using Methods in GUI Programs

You can call methods from other methods in a GUI application in the same way you can in a console application. Some special considerations when creating GUI applications include the following:

- Understanding methods that are automatically generated in the visual environment
- Appreciating scope in a GUI program
- Creating methods to be nonstatic when associated with a `Form`

### Understanding Methods That Are Automatically Generated in the Visual Environment

When you create GUI applications using the IDE, many methods are generated automatically. For example, when you place a `Button` named `okButton` on a `Form` in the IDE and double-click it, a method is generated with the following header:

```
private void okButton_Click(object sender, EventArgs e)
```

The method is `private`, which means it can be used only within its class, and it is `void`, meaning it does not return a value. You could change the access specifier to `public` and the method would still work, but usually you would have no reason to make such a change. If you change the return type (`void`), the program will not compile.

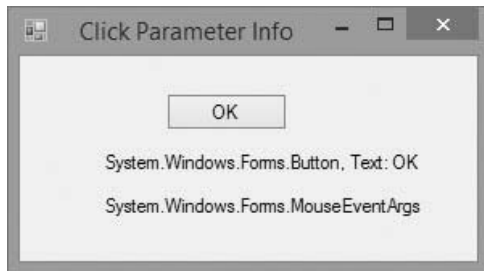
The format of the method name `okButton_Click()` is conventional for automatically created event handling methods. The name includes the control name, an underscore, and the event that causes the method to execute—in this case, a `Click` event. Although the C# convention is to begin method names with an uppercase letter, event-handling method names like `okButton_Click()` begin with a lowercase letter to echo the format of the control's name.

The parameters received by the `okButton_Click()` method arrive automatically when a user clicks the `okButton`. The parameter `sender` is the object that generated the event that caused the method to execute, and the parameter `e` contains information about the type of event. Figure 7-24 shows a `Click()` method in a GUI application that displays the `ToString()` values of these two parameters.

```
private void okButton_Click(object sender, EventArgs e)
{
    label11.Text = sender.ToString();
    label12.Text = e.ToString();
}
```

**Figure 7-24** The `okButton_Click()` method in the `ClickParameterInfo` program

Figure 7-25 shows the output after the button is clicked. The object that generated the event is the button that contains the text *OK*, and the type of event was generated by a mouse. You have seen many `Click()` methods in earlier chapters that had the same parameters, but you did not use them. As with any other method, you never are required to use the parameters passed in.



**Figure 7-25** Output generated by the `okButton_Click()` method in Figure 7-24

## Appreciating Scope in a GUI Program

When you create event-handling methods in a GUI application, you must constantly be aware of which variables and constants are needed by multiple methods. When you declare a variable or constant within a method, it is local to that method. If a variable or constant is needed by multiple event-handling methods—for example, by two different `Click()` methods—then the variables or constants in question must be defined outside the methods (but within the `Form` class) so that both `Click()` methods have access to them. When you write other methods, you can decide what arguments to pass in, but automatically generated event-handling methods have predefined sets of parameters, so you cannot be as flexible in using them as you can with other methods you write. In Chapter 9, “Using Classes and Objects,” you see many examples of variables declared within a class but outside methods; such variables are called *fields*.

## Creating Methods to be Nonstatic When Associated with a Form

In Chapter 1, you learned that the keyword `static` is used with `Main()` methods in console applications because the method is associated with its containing class and not with an object. When you create a GUI application and generate a method, such as a `Click()` method associated with a `Button` on a `Form`, the keyword `static` does not appear in the method header because the method is associated with an object—the `Form` from which the method-invoking events are sent. When you execute a GUI program using a `Form`, the program creates a `Form` object, and the methods associated with it are nonstatic. In the chapter “Using Classes and Objects,” you will learn much more about the differences between static and nonstatic methods. For now, omit the word `static` when you create methods that are intended to be called by other nonstatic methods in your GUI applications. For example, if you create a `Click()` method that responds to button clicks, it will be nonstatic by default because it is associated with an object that is a `Form`.



## TWO TRUTHS & A LIE

### Issues Using Methods in GUI Programs

1. An advantage to using the IDE to create GUI programs is that many methods you need are generated automatically.
2. As with any other methods, when you create a `Click()` method associated with a button in a GUI application, you do not need to reference the parameters within the method body.
3. If you want a `Click()` method to add two variables, you must pass them in as parameters.

The false statement is #3. Automatically generated event-handling methods have predefined sets of parameters, so you cannot be as flexible in using them as you can with other methods you write. Typically the variables used in a `Click()` method would be defined outside the method but inside the Form class where the `Click()` method resides.

## Chapter Summary

- A method is a series of statements that carry out a task. Any class can contain an unlimited number of methods. Object-oriented programs hide their methods' implementations.
- You write methods to make programs easier to understand and so that you can easily reuse them. In C#, a method must include a method declaration, an opening curly brace, a method body, and a closing curly brace. The method declaration defines the rules for using the method; it contains an optional declared accessibility, an optional `static` modifier, a return type for the method, an identifier, and an optional list of method parameters between parentheses.
- Some method calls require arguments. The items received by a method are parameters. When you write the declaration for a method that can receive a parameter, you need to include the type of parameter and a local identifier for it within the method declaration parentheses. A variable declared in a method header is a formal parameter, and an argument within a method call is an actual parameter.
- You can pass multiple arguments to a method by listing the arguments within the parentheses in the call to the method and separating them with commas. You can write a method to take any number of parameters. When you call the method, the arguments you

send to it must match in both number and type (allowing for type conversion) with the nondefault parameters listed in the method declaration.

- The return type for a method defines the type of value sent back to the calling method. It can be any type used in the C# programming language, which includes the basic built-in types `int`, `double`, `char`, and so on, as well as class types (including class types you create). A method also can return nothing, in which case the return type is `void`.
- You can pass an array as a parameter to a method. You indicate that a method parameter is an array by placing square brackets after the data type in the method's parameter list. Arrays, like all objects but unlike built-in types, are passed by reference; that is, the method receives the actual memory address of the array and has access to the actual values in the array elements.
- You might see different `Main()` method headers in other books or in programs written by others. Some programmers include an array of strings as a parameter to `Main()`, some programmers return an `int` from the `Main()` method, and some programmers give `Main()` methods public access.
- Special considerations exist for methods when you create GUI applications. You must understand the automatically generated methods in the visual environment, appreciate the differences in scope in GUI programs, and understand that many methods must be nonstatic when associated with a `Form`.

## Key Terms

A **method** is an encapsulated series of statements that carry out a task.

**Invoked** and **called** describe what one method has done to execute another.

A **calling method** calls another method.

A **called method** is invoked by another method.

**Implementation hiding** means keeping the details of a method's operations hidden.

A **client** is a method that uses another method.

A **black box** is any device you can use without knowing how it works internally.

A **multifile assembly** is a group of files containing methods that work together to create an application.

**Code bloat** describes unnecessarily long or repetitive program statements.

A **method declaration** is a **method header** or **method definition**.

A **method body** is a block of statements that carry out a method's work.

**Accessibility** for a method is a declaration that sets limits as to whether and how other methods can use it.

**Public access** is a level of method accessibility that allows unlimited access to a method by any class.

**Private access** is a level of method accessibility that limits method access to the containing class.

A **fully qualified** name is one that includes the class name.

A **static** method is called without an object reference. It can be called from any class using its class name, a dot, and the method name. Within its own class, a static method can be called by just using its name.

A **nonstatic** method call requires an object reference.

A **return type** indicates what kind of value a method will return to any other method that calls it.

A **method's type** is its return type.

A **parameter to a method** is a variable that holds data passed to a method when it is called.

A **parameter list** consists of the data types and parameter names that appear between parentheses in a method header. The list might hold only one item or multiple items separated with commas.

A variable's **scope** is the segment of program code where it can be used.

A **local** variable is one that is declared in the current method.

A **formal parameter** is a parameter within a method header that accepts a value.

**Actual parameters** are arguments within a method call.

A **return statement** causes a value to be sent back from a method to its calling method.

**Nested method calls** are method calls placed inside other method calls.

**Passed by reference** describes the state of an argument passed to a method when the method receives its memory address.

## Review Questions

- At most, a class can contain \_\_\_\_\_ method(s).
  - 0
  - 1
  - 2
  - any number of
- Which of the following is a good reason for creating methods within a program?
  - Methods are easily reusable.
  - Because all methods must be stored in the same class, they are easy to find.
  - The `Main()` method becomes more detailed.
  - All of these are true.

3. In C#, a method must include all of the following *except* a \_\_\_\_\_.
  - a. return type
  - b. access modifier
  - c. body
  - d. closing curly brace
4. A method declaration might contain \_\_\_\_\_.
  - a. declared accessibility
  - b. a nonstatic modifier
  - c. multiple return types
  - d. parameters separated by dots
5. Every method declaration must contain \_\_\_\_\_.
  - a. a statement of purpose
  - b. declared accessibility
  - c. the **static** modifier
  - d. a return type
6. If you want to create a method that other methods in other classes can access without limitations, you declare the method to be \_\_\_\_\_.
  - a. unlimited
  - b. public
  - c. shared
  - d. unrestricted
7. If you use the keyword modifier **static** in a method header, you indicate that the method \_\_\_\_\_.
  - a. cannot be copied
  - b. can be called only once
  - c. cannot require parameters
  - d. is called without an object reference
8. A method's type is also its \_\_\_\_\_.
  - a. return type
  - b. accessibility
  - c. parameter type
  - d. scope
9. When you use a method, you do not need to know how it operates internally. This feature is called \_\_\_\_\_.
  - a. scope management
  - b. selective ignorance
  - c. implementation hiding
  - d. privacy
10. When you write the method declaration for a method that can receive a parameter, you need to include all of the following items *except* \_\_\_\_\_.
  - a. a pair of parentheses
  - b. the type of the parameter
  - c. a local name for the parameter
  - d. an initial value for the parameter

11. Suppose you have declared a variable as `int myAge = 21;`. Which of the following is a legal call to a method with the declaration `private static void AMethod(int num)?`
- a. `AMethod(int 55);`
  - b. `AMethod(myAge);`
  - c. `AMethod(int myAge);`
  - d. `AMethod();`
12. Suppose you have declared a method named `private static void CalculatePay(double rate)`. When a method calls the `CalculatePay()` method, the calling method \_\_\_\_\_.
- a. must contain a declared `double` named `rate`
  - b. might contain a declared `double` named `rate`
  - c. cannot contain a declared `double` named `rate`
  - d. cannot contain any declared `double` variables
13. In the method call `PrintTheData(double salary);`, `salary` is the \_\_\_\_\_ parameter.
- a. formal
  - b. actual
  - c. proposed
  - d. preferred
14. A program contains the method call `PrintTheData(salary);`. In the method definition, the name of the formal parameter must be \_\_\_\_\_.
- a. `salary`
  - b. any legal identifier other than `salary`
  - c. any legal identifier
  - d. omitted
15. What is a correct declaration for a method that receives two `double` arguments, calculates and displays the difference between them, and returns nothing?
- a. `private static void CalcDifference(double price1, price2)`
  - b. `private static void CalcDifference(double price1, double price2)`
  - c. Both of these are correct.
  - d. None of these are correct.

16. What is a correct declaration for a method that receives two `double` arguments and sums them but does not return anything?
- `private static void CalcSum(double firstValue, double secondValue)`
  - `private static void CalcSum(double price1, double price2)`
  - Both of these are correct.
  - None of these are correct.
17. A method is declared as `private static double CalcPay(int hoursWorked)`. Suppose you write a `Main()` method in the same class that contains the declarations `int hours = 35;` and `double pay;`. Which of the following represents a correct way to call the `CalcPay()` method from the `Main()` method?
- `hours = CalcPay();`
  - `hours = Main.CalcPay();`
  - `pay = CalcPay(hoursWorked);`
  - `pay = CalcPay(hours);`
18. Suppose the value of `isRateOK()` is `true`, and the value of `isQuantityOK()` is `false`. When you evaluate the expression `isRateOK() || isQuantityOK()`, which of the following is true?
- Only the method `isRateOK()` executes.
  - Only the method `isQuantityOK()` executes.
  - Both methods execute.
  - Neither method executes.
19. Suppose the value of `isRateOK()` is `true`, and the value of `isQuantityOK()` is `false`. When you evaluate the expression `isRateOK() && isQuantityOK()`, which of the following is true?
- Only the method `isRateOK()` executes.
  - Only the method `isQuantityOK()` executes.
  - Both methods execute.
  - Neither method executes.
20. When an array is passed to a method, the method has access to the array's memory address. This means an array is passed by \_\_\_\_\_.
- reference
  - value
  - alias
  - orientation

## Exercises



### Programming Exercises

307

1. Create a C# statement that uses each of the following built-in methods you have used in previous chapters, then make an intelligent guess about the return type and parameter list for the method used in each statement you created.
  - a. `Console.WriteLine()`
  - b. `String.Equals()`
  - c. `String.Compare()`
  - d. `Convert.ToInt32()`
  - e. `Convert.ToChar()`
  - f. `Array.Sort()`
2. Create a program named **SalesLetter** whose `Main()` method uses several `WriteLine()` calls to display a sales letter to prospective clients for a business of your choice. Display your contact information at least three times in the letter. The contact information should contain at least three lines with data such as land line phone number, cell number, email address, and so on. Each time you want to display the contact information, call a method named `DisplayContactInfo()`.
3. Create a program named **PaintingEstimate** whose `Main()` method prompts a user for length and width of a room in feet. Create a method that accepts the values and then computes the cost of painting the room, assuming the room is rectangular and has four full walls and 9-foot ceilings. The price of the job is \$6 per square foot. Return the price to the `Main()` method, and display it.
4. Create an application named **ConvertQuartsToLiters** whose `Main()` method prompts a user for a number of quarts, passes the value to a method that converts the value to liters, and then returns the value to the `Main()` method where it is displayed. A quart is 0.966353 liters.
5. Create a program named **FortuneTeller** whose `Main()` method contains an array of at least six strings with fortune-telling phrases such as *I see a tall, dark stranger in your future*. The program randomly selects two different fortunes and passes them to a method that displays them.
6. Create an application named **Multiplication** whose `Main()` method asks the user to input an integer and then calls a method named `DisplayMultiplicationTable()`, which displays the results of multiplying the integer by each of the numbers 2 through 10.
7. In Chapter 4, you wrote a program named **Admission** for a college admissions office in which the user enters a numeric high school grade point average and an admission test score. The program displays *Accept* or *Reject* based on those values. Now, create a modified program named **AdmissionModularized** in which the grade point average and test score are passed to a method that returns a string containing *Accept* or *Reject*.

8. Write a program named **CountVowelsModularized** that passes a string to a method that returns the number of vowels in the string.
9. Create an application named **Desks** that computes the price of a desk and whose **Main()** method calls the following methods:
  - A method to accept the number of drawers in the desk as input from the keyboard. This method returns the number of drawers to the **Main()** method.
  - A method to accept as input and return the type of wood, *m* for mahogany, *o* for oak, or *p* for pine.
  - A method that accepts the number of drawers and wood type, and calculates the cost of the desk based on the following:
    - Pine desks are \$100.
    - Oak desks are \$140.
    - All other woods are \$180.
    - A \$30 surcharge is added for each drawer.
  - This method returns the cost to the **Main()** method.
  - A method to display all the details and the final price.
10. Create a program named **FlexibleArrayMethod** that declares at least three integer arrays of different sizes. In turn, pass each array to a method that displays all the integers in each array and their sum.



## Debugging Exercises

1. Each of the following files in the *Chapter.07* folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem, and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, *DebugSeven1.cs* will become *FixedDebugSeven1.cs*.
  - a. *DebugSeven1.cs*
  - b. *DebugSeven2.cs*
  - c. *DebugSeven3.cs*
  - d. *DebugSeven4.cs*





## Case Problems

1. In Chapter 6, you continued to modify the **GreenvilleRevenue** program. Now, modify the program so that the major functions appear in the following individual methods:
  - A method that gets and returns a valid number of contestants and is called twice—once for last year's number of contestants and once for this year's value
  - A method that accepts the number of contestants this year and last year and displays one of the three messages that describes the relationship between the two contestant values
  - A method that fills the array of competitors and their talent codes
  - A method that continuously prompts for talent codes and displays contestants with the corresponding talent until a sentinel value is entered
2. In previous chapters, you continued to modify the **MarshallsRevenue** program. Now, modify the program so that the major functions appear in the following individual methods:
  - A method that prompts for and returns the month
  - A method that prompts for and returns the number of murals scheduled and is called twice—once for interior murals and once for exterior murals
  - A method that accepts the number of interior and exterior murals scheduled, accepts the month they are scheduled, displays the interior and exterior prices, and then returns the total expected revenue
  - A method that fills an array with customer names and mural codes and is called twice—once to fill the array of interior murals and once to fill the array of exterior murals
  - A method that continuously prompts for mural codes and displays jobs of the corresponding type until a sentinel value is entered

