CHAPTER 8

Advanced Method Concepts

In this chapter you will:

- Learn about parameter types and review mandatory value parameters
- Use `ref` parameters, `out` parameters, and parameter arrays
- Overload methods
- Learn how to avoid ambiguous methods
- Use optional parameters

In the last chapter you learned to call methods, pass arguments to them, and receive values from them. In this chapter you will expand your method-handling skills to include more sophisticated techniques, including using reference, output, and optional parameters and overloading methods. Understanding how to manipulate methods is crucial when you are working on large, professional, real-world projects. With methods, you can more easily coordinate your work with that of other programmers.

Understanding Parameter Types

In C#, an argument can be passed to a method's parameter by value or by reference. In the previous chapter, you passed arguments of simple data types (like `int` and `double`) by value, and you passed arrays by reference. Parameters also can be classified as mandatory or optional. When you use a **mandatory parameter**, an argument for it is required in every method call. An **optional parameter** to a method is one for which a default value is automatically supplied if you do not explicitly send one as an argument. All the methods that you have seen with simple and array parameters have used mandatory parameters.

Mandatory parameters also include:

- Reference parameters
- Output parameters
- Parameter arrays

C# includes only one type of optional parameter:

- Value parameters, when they are declared *with* default values

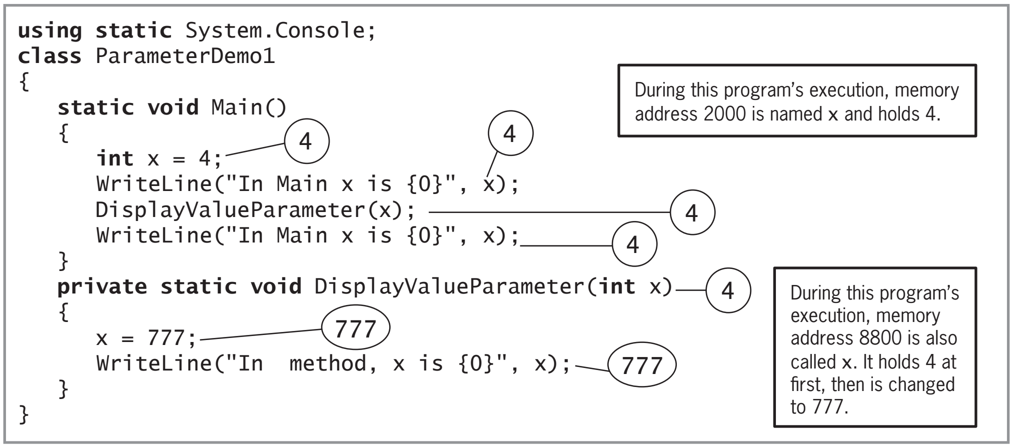
Using Mandatory Value Parameters

So far, all of the method parameters you have created have been mandatory, and all except arrays have been value parameters. When you use a **value parameter** in a method header, you indicate the parameter's type and name, and the method receives a copy of the value passed to it. A variable that is used as an argument to a method with a value parameter must have a value assigned to it. If it does not, the program will not compile.

The value of a method's value parameter is stored at a different memory address than the variable used as the argument in the method call. In other words, the actual parameter (the argument in the calling method) and the formal parameter (the parameter in the method header) refer to two separate memory locations, and the called method receives a copy of the sent value. Changes to value parameters never affect the original arguments in calling methods.

A popular advertising campaign declares, "What happens in Vegas, stays in Vegas." The same is true of value parameters within a method—changes to them do not persist outside the method.

Figure 8-1 shows a program that declares a variable named `x`; the figure assumes that `x` is stored at memory address 2000. The value 4 is assigned to `x` and then displayed. Then `x` is passed to a method that accepts a value parameter. The method declares its own local parameter named `x`, which receives the value 4. This local variable is at a different memory address; for example, assume that it is 8800. The method assigns a new value, 777, to the local variable and displays it. When control returns to the `Main()` method, the value of `x` is accessed from memory location 2000 and remains 4.

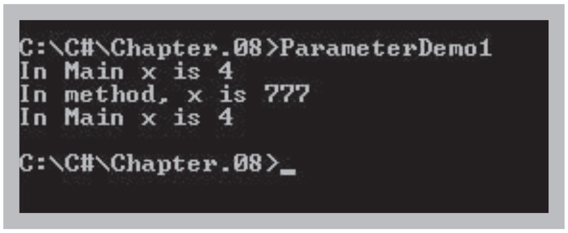


```
using static System.Console;
class ParameterDemo1
{
   static void Main()
   {
      int x = 4;
      WriteLine("In Main x is {0}", x);
      DisplayValueParameter(x);
      WriteLine("In Main x is {0}", x);
   }
   private static void DisplayValueParameter(int x)
   {
      x = 777;
      WriteLine("In  method, x is {0}", x);
   }
}
```

4
4
4
4
4
777
777
During this program's execution, memory address 2000 is named x and holds 4.
During this program's execution, memory address 8800 is also called x. It holds 4 at first, then is changed to 777.

Figure 8-1 Program calling a method with a value parameter

Changing the value of `x` within the `DisplayValueParameter()` method has no effect on `x` in the `Main()` method. Even though both methods contain a variable named `x`, they represent two separate variables, each with its own memory location. Programmers say that a value parameter is used for "in" parameter passing—values for "in" parameters go into a method, but modifications to them do not come "out." In the program in Figure 8-1, it makes no difference whether you use the name `x` as the `Main()` method's actual parameter or use some other name. In either case, the parameter received by the `DisplayValueParameter()` method occupies a separate memory location. Figure 8-2 shows the output of the program in Figure 8-1.



```
C:\C#\Chapter.08>ParameterDemo1
In Main x is 4
In method, x is 777
In Main x is 4

C:\C#\Chapter.08>_
```

Figure 8-2 Output of the ParameterDemo1 program

TWO TRUTHS & A LIE

Understanding Parameter Types

1. When you call a method that has a mandatory parameter, you must send an argument.
2. When you call a method that has an optional parameter, you do not have to send an argument, but you can.
3. When you call a method with a mandatory value parameter, you must pass an argument, but if it is a variable, you do not have to have initialized it.

The false statement is #3. A mandatory value parameter must receive an initialized argument.

Using Reference Parameters, Output Parameters, and Parameter Arrays

On occasion, you might want a method to be able to alter a value you pass to it. In that case, you can use a reference parameter, an output parameter, or a parameter array. Each **reference parameter**, **output parameter**, and **parameter array** represents a memory address that is passed to a method, allowing the method to alter the original variable. Reference parameters, output parameters, and parameter arrays differ as follows:

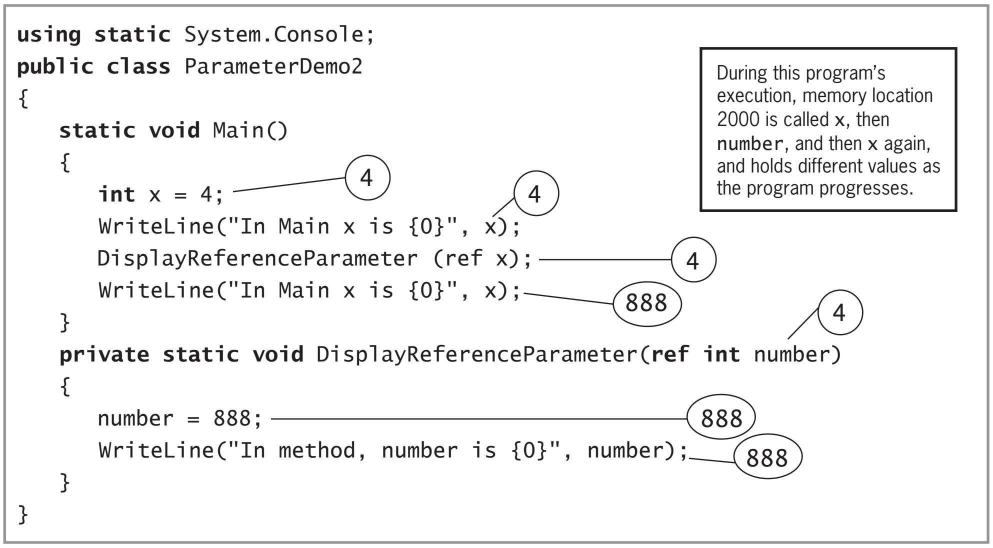
- When you declare a reference parameter in a method header, the argument used to call the method must have been assigned a value.
- When you declare an output parameter, the argument used in the call need not contain an original, assigned value. However, an output parameter must be assigned a value before the method ends.
- When you declare a parameter array, the argument used in the call need not contain any original, assigned values, and the parameter array need not be assigned any values within the method.

Reference parameters, output parameters, and parameter arrays do not occupy their own memory locations. Rather, they act as aliases, or pseudonyms (other names), for the same memory location occupied by the values passed to them. You use the keyword `ref` as a modifier to indicate a reference parameter, the keyword `out` as a modifier to indicate an output parameter, and the keyword `params` to indicate a parameter array.

Using an alias for a variable is similar to using an alias for a person. Jane Doe might be known as "Ms. Doe" at work but "Sissy" at home. Both names refer to the same person.

Using a `ref` Parameter

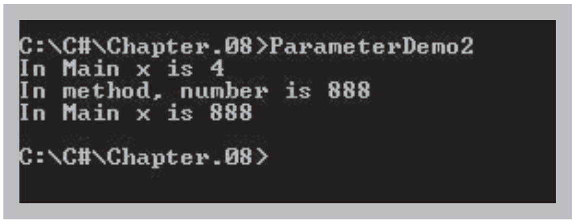
Figure 8-3 shows a `Main()` method that calls a `DisplayReferenceParameter()` method. The `Main()` method declares and initializes a variable, displays its value, and then passes the variable to the method. The modifier `ref` precedes both the variable name in the method call and the parameter declaration in the method header. The method's parameter `number` refers to the memory address of `x`, making `number` an alias for `x`. When `DisplayReferenceParameter()` changes the value of `number`, the change persists in the `x` variable within `Main()`. Figure 8-4 shows the output of the program.



```
using static System.Console;
public class ParameterDemo2
{
   static void Main()
   {
      int x = 4;
      WriteLine("In Main x is {0}", x);
      DisplayReferenceParameter (ref x);
      WriteLine("In Main x is {0}", x);
   }
   private static void DisplayReferenceParameter(ref int number)
   {
      number = 888;
      WriteLine("In method, number is {0}", number);
   }
}
```

During this program's execution, memory location 2000 is called `x`, then `number`, and then `x` again, and holds different values as the program progresses.

Figure 8-3 Program calling a method with a reference parameter



```
C:\C#\Chapter.08>ParameterDemo2
In Main x is 4
In method, number is 888
In Main x is 888

C:\C#\Chapter.08>
```

Figure 8-4 Output of the ParameterDemo2 program

In the header for the `DisplayReferenceParameter()` method, it makes no difference whether you use the same name as the `Main()` method's passed variable (`x`) or some other name, such as `number`. In either case, the passed and received variables occupy the same memory location—the address of one is the address of the other.

Using an `out` Parameter

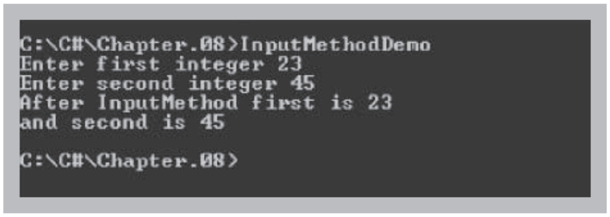
When you use a reference parameter, any passed variable must have an assigned value. Using an output parameter is convenient when the passed variable doesn't have a value yet. For example, the program in Figure 8-5 uses `InputMethod()` to obtain values for two parameters. The arguments that are sent in the shaded statement get their values from the called method, so it makes sense to provide them with no values going in. Instead, they acquire values in the method and retain the values coming out. Figure 8-6 shows a typical execution of the program.

```
using System;
using static System.Console;
class InputMethodDemo
{
   static void Main()
   {
      int first, second;
      InputMethod(out first, out second);
      WriteLine("After InputMethod first is {0}", first);
      WriteLine("and second is {0}", second);
   }
   private static void InputMethod(out int one, out int two)
   {
      string s1, s2;
      Write("Enter first integer ");
      s1 = ReadLine();
      Write("Enter second integer ");
      s2 = ReadLine();
      one = Convert.ToInt32(s1);
      two = Convert.ToInt32(s2);
   }
}
```

Notice the keyword `out`.

Notice the keyword `out`.

Figure 8-5 The `InputMethodDemo` program



```
C:\C#\Chapter.08>InputMethodDemo
Enter first integer 23
Enter second integer 45
After InputMethod first is 23
and second is 45

C:\C#\Chapter.08>
```

Figure 8-6 Output of the `InputMethodDemo` program

It might be helpful to remember:

- If you will always have an initial value, use `ref`.
- If you do not have an initial value or you do not know whether you will have an initial value, use `out`. (For example, in an interactive program, a user might not provide a value for a variable, so you don't know whether the variable has an initial value.)

In summary, when you need a method to share a single value from a calling method, you have two options:

- Use a return type. You can send an argument to a method that accepts a value parameter, alter the local version of the variable within the method, return the altered value, and assign the return value to the original variable back in the calling method. The drawback to this approach is that a method can have only a single return type and can return only one value at most.
- Pass by reference. You can send an argument to a method that accepts a reference or output parameter and alter the value at the original memory location within the method. A major advantage to using reference or output parameters exists when you want a method to change multiple variables. A method can have only a single return type and can return only one value at most, but by using reference or output parameters to a method, you can change multiple values. However, a major disadvantage to using reference and output parameters is that they allow multiple methods to have access to the same data, weakening the "black box" paradigm that is so important to object-oriented methodology. You should never use `ref` or `out` parameters to avoid having to return a value, but you should understand them so you can use them if required.

Watch the video *Using `ref` and `out` Parameters*.

As with simple parameters, you can use `out` or `ref` when passing an array to a method. You do so when you want to declare the array in the calling method, but use the `new` operator to allocate memory for it in the called method.

Using a Built-in Method That Has an `out` Parameter

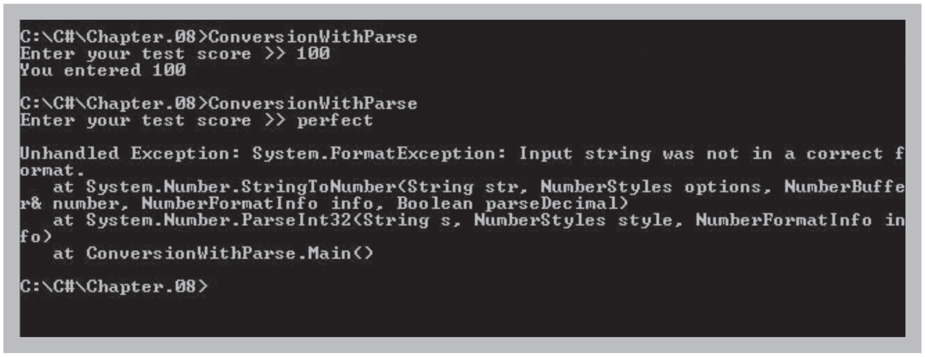
In Chapter 2, you learned two ways to convert a string to a number—you can use a `Convert` class method or a `Parse()` method. For example, each of the following statements accepts a string and converts it to an integer named `score`:

```
int score = Convert.ToInt32(ReadLine());
int score = int.Parse(ReadLine());
```

With either of these statements, if the value accepted by `ReadLine()` cannot be converted to an integer (for example, because it contains a decimal point or an alphabetic character), the program abruptly stops running. Consider the simple program in Figure 8-7. The program prompts the user for a score and displays it. Figure 8-8 shows two executions of the program. In the first execution, the user enters a valid integer and everything goes smoothly. In the second execution, the user enters alphabetic characters, the program stops, and a series of error messages is displayed.

```
using static System.Console;
class ConversionWithParse
{
   static void Main()
   {
      string entryString;
      int score;
      Write("Enter your test score >> ");
      entryString = ReadLine();
      score = int.Parse(entryString);
      WriteLine("You entered {0}", score);
   }
}
```

Figure 8-7 The ConversionWithParse program



```
C:\C#\Chapter.08>ConversionWithParse
Enter your test score >> 100
You entered 100

C:\C#\Chapter.08>ConversionWithParse
Enter your test score >> perfect

Unhandled Exception: System.FormatException: Input string was not in a correct f
ormat.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffe
r& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo in
fo)
   at ConversionWithParse.Main()

C:\C#\Chapter.08>
```

Figure 8-8 Two typical executions of the `ConversionWithParse` program

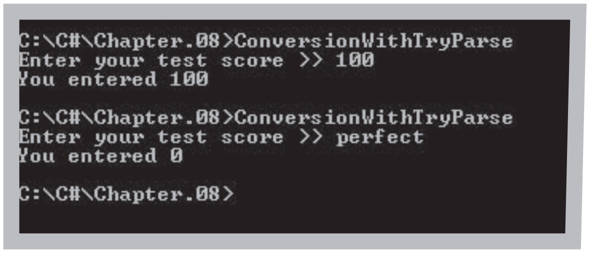
The messages in Figure 8-8 indicate that the program contains an unhandled exception because *Input string was not in a correct format*. An exception is a program error; you will learn much more about exceptions in Chapter 11, "Exception Handling." For now, however, you can handle data conversion exceptions by the `TryParse()` method. Unlike the `Parse()` method, `TryParse()` accepts an `out` parameter. The `TryParse()` method converts a string to the correct data type and stores the result in a passed variable if possible. If conversion is not possible, the method assigns 0 to the variable.

Figure 8-9 contains a program that uses the `int.TryParse()` method to convert `entryString` to an integer that is assigned to `score`. The only differences from the program in Figure 8-7 are the shaded change to the class name and the shaded call to the `int.TryParse()` method.

The TryParse() method accepts two parameters: the string to be converted and the variable where the result is stored. The keyword out is required with the second parameter because the method receives its address and changes the value. Figure 8-10 shows two typical executions of the ConversionWithTryParse program. When the user's input cannot be converted correctly, the out parameter is assigned 0. Whether or not the user's input is in the correct format, the program continues and displays output instead of error messages.

```
using static System.Console;
class ConversionWithTryParse
{
   static void Main()
   {
      string entryString;
      int score;
      Write("Enter your test score >> ");
      entryString = ReadLine();
      int.TryParse(entryString, out score);
      WriteLine("You entered {0}", score);
   }
}
```

Figure 8-9 The ConversionWithTryParse program



```
C:\C#\Chapter.08>ConversionWithTryParse
Enter your test score >> 100
You entered 100

C:\C#\Chapter.08>ConversionWithTryParse
Enter your test score >> perfect
You entered 0

C:\C#\Chapter.08>
```

Figure 8-10 Two typical executions of the ConversionWithTryParse program

The TryParse() method requires the receiving variable to be an out parameter for two reasons:

- The argument does not have an assigned value in the calling method.
- The method returns a Boolean value that indicates whether the conversion was successful and so cannot return the score value too.

Suppose that you do not want a score to be assigned 0 if the conversion fails, because 0 is a legitimate score. Instead, you want to assign a –1 if the conversion fails. In that case you can use a statement similar to the following:

```
if(!int.TryParse(entryString, out score))
   score = -1;
```

In this example, if `entryString`'s conversion is successful, `score` holds the converted value; otherwise, `score` holds –1.

As another example, you might consider code that includes a loop that continues until the entered value is in the correct format, such as the following:

```
Write("Enter your test score >> ");
entryString = ReadLine();
while(!int.TryParse(entryString, out score))
{
   WriteLine("Input data was not formatted correctly");
   Write("Please enter score again >> ");
   entryString = ReadLine();
}
```

In this example, the loop continues until the `TryParse()` method returns `true`.

C# also provides `char.TryParse()`, `double.TryParse()`, and `decimal.TryParse()` methods that work in the same way—each converts a first parameter string and assigns its value to the second parameter variable, returning `true` or `false` based on the success of the conversion.

Using Parameter Arrays

When you don't know how many arguments of the same data type might eventually be sent to a method, you can declare a parameter array—a local array declared within the method header using the keyword `params`. Such a method accepts any number of elements that are all the same data type.

For example, a method with the following header accepts an array of strings:

```
private static void DisplayStrings(params string[] people)
```

In the call to this method, you can use any number of strings as actual parameters; within the method, they will be treated as an array.

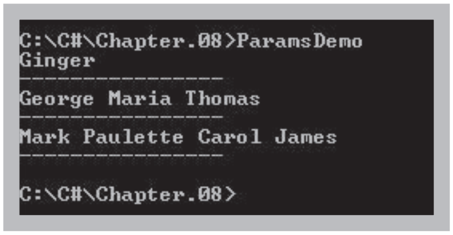
When a method header uses the `params` keyword, the following two restrictions apply:

- Only one `params` keyword is permitted in a method declaration.
- If a method declares multiple parameters, the `params`-qualified parameter must be the last one in the list.

Figure 8-11 shows a program that calls `DisplayStrings()` three times—once with one string argument, once with three string arguments, and once with an array of strings. In each case, the method works correctly, treating the passed strings as an array and displaying them appropriately. Figure 8-12 shows the output.

```
using static System.Console;
class ParamsDemo
{
   static void Main()
   {
      string[] names = {"Mark", "Paulette", "Carol", "James"};
      DisplayStrings("Ginger");
      DisplayStrings("George", "Maria", "Thomas");
      DisplayStrings(names);
   }
   private static void DisplayStrings(params string[] people)
   {
      foreach(string person in people)
         Write("{0} ", person);
      WriteLine("\n--------------");
   }
}
```

Figure 8-11 The `ParamsDemo` program



```
C:\C#\Chapter.08>ParamsDemo
Ginger
----------------
George Maria Thomas
----------------
Mark Paulette Carol James
----------------

C:\C#\Chapter.08>
```

Figure 8-12 Output of the `ParamsDemo` program

You could create an even more flexible method by using a method header such as `Display(params Object[] things)`. Then the passed parameters could be any type—strings, integers, other classes, and so on. The method could be implemented as follows:

```
private static void Display(params Object[] things)
{
   foreach(Object obj in things)
      Write("{0} ", obj);
   WriteLine("\n--------------");
}
```

All data types are `Objects`; you will learn more about the `Object` class in the chapters "Using Classes and Objects" and "Introduction to Inheritance."

TWO TRUTHS & A LIE

Using Reference Parameters, Output Parameters, and Parameter Arrays

1. Both reference and output parameters represent memory addresses that are passed to a method, allowing the method to alter the original variables.
2. When you declare a reference parameter in a method header, the parameter must not have been assigned a value.
3. When you use an output parameter, it need not contain an original, assigned value when the method is called, but it must receive a value before the method ends.

The false statement is #2. When you declare a reference parameter in a method header, the parameter must have been assigned a value.

You Do It

Using Reference Parameters

You use reference parameters when you want a method to have access to the memory address of arguments in a calling method. For example, suppose that you have two values and you want to exchange them (or swap them), making each equal to the value of the other. Because you want to change two values, a method that accepts copies of arguments will not work—a method can return one value at most. Therefore, you can use reference parameters to provide your method with the actual addresses of the values you want to change.

1. Open your editor and begin a program named **SwapProgram** as follows:

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

2. Declare two integers, and display their values. Call the `Swap()` method, and pass in the addresses of the two variables to swap. Because the parameters already have assigned values, and because you want to alter those values

(continues)

(continued)

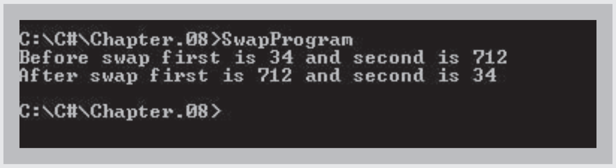
in `Main()`, you can use reference parameters. After the method call, display the two values again. Add the closing curly brace for the `Main()` method.

```
   int first = 34, second = 712;
   Write("Before swap first is {0}", first);
   WriteLine(" and second is {0}", second);
   Swap(ref first, ref second);
   Write("After swap first is {0}", first);
   WriteLine(" and second is {0}", second);
}
```

3. Create the `Swap()` method as shown. You can swap two values by storing the first value in a temporary variable, then assigning the second value to the first variable. At this point, both variables hold the value originally held by the second variable. When you assign the temporary variable's value to the second variable, the two values are reversed.

```
private static void Swap(ref int one, ref int two)
{
   int temp;
   temp = one;
   one = two;
   two = temp;
}
```

4. Add the closing curly brace for the class. Save the program, and then compile and execute it. Figure 8-13 shows the output.



```
C:\C#\Chapter.08>SwapProgram
Before swap first is 34 and second is 712
After swap first is 712 and second is 34

C:\C#\Chapter.08>
```

Figure 8-13 Output of the SwapProgram program

You might want to use a module like `Swap()` as part of a larger program in which you verify, for example, that a higher value is displayed before a lower one; you would include the call to `Swap()` as part of a decision whose body executes only when a first value is less than a second one.

Overloading Methods

Overloading involves the ability to write multiple versions of a method using the same method name. When you make a purchase at a store, you might use one of a variety of "pay" methods they accept, for example, cash, a credit card, or a check. The "pay" method works differently depending on the type of currency sent to it, but all of the transactions are called “pay.” When you overload a C# method, you write multiple method implementations with the same name but different parameter lists. A method’s name and parameter list constitute the method’s **signature**.

In this book you have seen the `WriteLine()` method used with a string parameter and a numeric parameter. You can also use it with no parameter to output a blank line. You have also seen the method used with several parameters when you have used a format string along with several variables. Therefore, you know `WriteLine()` is an overloaded method.

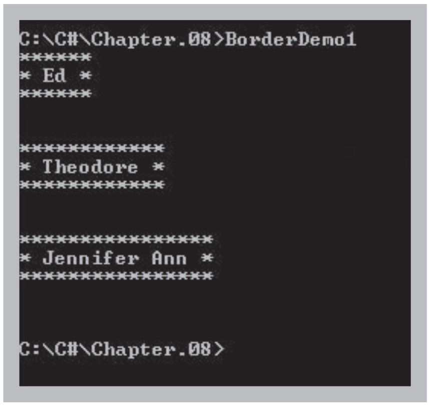
Some C# operators are overloaded. For example, a + between two numeric values indicates addition, but a single + to the left of a value means the value is positive. The + sign has different meanings based on the operands used with it. In the chapter “Using Classes and Objects,” you will learn how to overload operators to make them mean what you want with your own classes.

The compiler understands which method to use based on the arguments you use in a method call. For example, suppose that you create a method to display a `string` surrounded by a border. The method receives a `string` and uses the `string Length` property to determine how many asterisks to use to construct a border around the `string`. Figure 8-14 shows a program that contains the method.

```
using static System.Console;
class BorderDemo1
{
   static void Main()
   {
      DisplayWithBorder("Ed");
      DisplayWithBorder("Theodore");
      DisplayWithBorder("Jennifer Ann");
   }
   private static void DisplayWithBorder(string word)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = "*";
      int size = word.Length + EXTRA_STARS;
      int x;
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine();
      WriteLine(SYMBOL + " " + word + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine("\n\n");
   }
}
```

Figure 8-14 The `BorderDemo1` program

When the `Main()` method calls the `DisplayWithBorder()` method in the program in Figure 8-14 and passes a `string` value, the method calculates a size as the length of the string plus 4, and then draws that many symbols on a single line. The method then displays a symbol, a space, the string, another space, and another symbol on the next line.



```
C:\C#\Chapter.08>BorderDemo1
******
* Ed *
******


************
* Theodore *
************


****************
* Jennifer Ann *
****************



C:\C#\Chapter.08>
```

Figure 8-15 Output of the BorderDemo1 program

Suppose that you are so pleased with the output of the `DisplayWithBorder()` method that you want to use something similar to display your company's weekly sales goal figure. The problem is that the weekly sales goal amount is stored as an integer, and so it cannot be passed to the existing method. You can take one of several approaches:

- You can convert the integer sales goal to a `string` and use the existing method. This is an acceptable approach, but it requires that you remember to write an extra step in any program in which you display an integer using the border.
- You can create a new method with a unique name such as `DisplayWithBorderUsingInt()` and use it to accept an integer parameter. The drawback to this approach is that you must remember different method names when you use different data types.
- You can overload the `DisplayWithBorder()` method. Overloading methods involves writing multiple methods with the same name but with different parameter types. For example, in addition to the `DisplayWithBorder()` method shown in Figure 8-14, you could use the method shown in Figure 8-16.

```
private static void DisplayWithBorder(int number)
{
   const int EXTRA_STARS = 4;
   const string SYMBOL = "*";
   int size = EXTRA_STARS + 1;
   int leftOver = number;
   int x;
   while(leftOver >= 10)
   {
      leftOver = leftOver / 10;
      ++size;
   }
   for(x = 0; x < size; ++x)
      Write(SYMBOL);
   WriteLine();
   WriteLine(SYMBOL + " " + number + " " + SYMBOL);
   for(x = 0; x < size; ++x)
      Write(SYMBOL);
   WriteLine("\n\n");
}
```

Figure 8-16 The `DisplayWithBorder()` method with an integer parameter

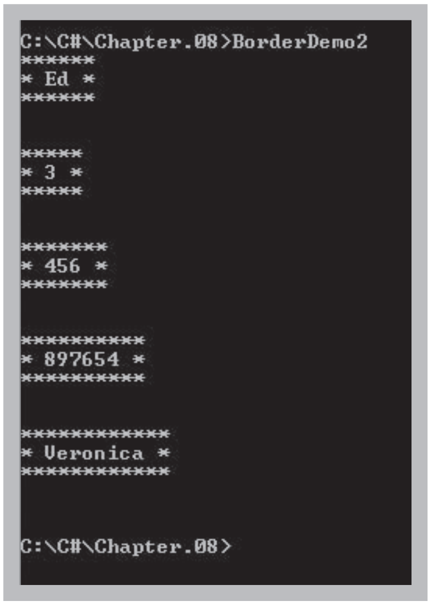
In the version of the `DisplayWithBorder()` method in Figure 8-16, the parameter is an `int`. To determine how many asterisks to display, the method initializes size to the number of extra stars in the display (in this case 4), plus one more. It then determines the number of asterisks to display in the border by repeatedly dividing the parameter by 10 and adding the result to `size`. For example, when the argument to the method is 456, `leftOver` is initialized to 456. Because it is at least 10, it is divided by 10, giving 45, and `size` is increased from 5 to 6. Then 45 is divided by 10, giving 4, and `size` is increased to 7. Because 4 is not at least 10, the loop ends, and the program has determined that the top and bottom borders of the box surrounding *456* require seven stars each. The rest of the method executes like the original version that accepts a `string` parameter.

The `DisplayWithBorder()` method does not quite work correctly if a negative integer is passed to it because the negative sign occupies an additional display space. To rectify the problem, you could modify the method to add an extra symbol to the border when a negative argument is passed in, or you could force all negative numbers to be their positive equivalent.

If both versions of `DisplayWithBorder()` are included in a program and you call the method using a `string`, as in `DisplayWithBorder("Ed")`, the first version of the method shown in Figure 8-14 executes. If you use an integer as the argument in the call to `DisplayWithBorder()`, as in `DisplayWithBorder(456)`, then the method shown in Figure 8-16 executes. Figure 8-17 shows a program that demonstrates several method calls, and Figure 8-18 shows the output.

```
using static System.Console;
class BorderDemo2
{
   static void Main()
   {
      DisplayWithBorder("Ed");
      DisplayWithBorder(3);
      DisplayWithBorder(456);
      DisplayWithBorder(897654);
      DisplayWithBorder("Veronica");
   }
   private static void DisplayWithBorder(string word)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = '*';
      int size = word.Length + EXTRA_STARS;
      int x;
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine();
      WriteLine(SYMBOL + " " + word + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine("\n\n");
   }
   private static void DisplayWithBorder(int number)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = "*";
      int size = EXTRA_STARS + 1;
      int leftOver = number;
      int x;
      while(leftOver >= 10)
      {
         leftOver = leftOver / 10;
         ++size;
      }
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine();
      WriteLine(SYMBOL + " " + number + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Write(SYMBOL);
      WriteLine("\n\n");
   }
}
```

Figure 8-17 The BorderDemo2 program



```
C:\C#\Chapter.08>BorderDemo2
******
* Ed *
******


*****
* 3 *
*****


*******
* 456 *
*******


**********
* 897654 *
**********


************
* Veronica *
************


C:\C#\Chapter.08>
```

Figure 8-18 Output of the `BorderDemo2` program

Methods are overloaded correctly when they have the same identifier but their parameter lists are different. Parameter lists differ when the number and order of types within the lists are unique. For example, you could write several methods with the same identifier, and one method could accept an `int`, another two `int`s, and another three `int`s. A fourth method could accept an `int` followed by a `double`, and another could accept a `double` followed by an `int`. Yet another version could accept no parameters. The parameter identifiers in overloaded methods do not matter, and neither do the return types of the methods. The only two requirements to overload methods are the same identifier and different parameter lists.

Instead of overloading methods, you can choose to use methods with different names to accept the diverse data types, and you can place a decision within your program to determine which version of the method to call. However, it is more convenient to use one method name and then let the compiler determine which method to use. Overloading a method also makes it more convenient for other programmers to use your method in the future. Usually you do not create overloaded methods so that a single program can use all the versions. More often, you create overloaded methods so different programs can use the version most appropriate to the task at hand. Frequently, you create overloaded methods in your classes not because you need them immediately, but because you know client programs might need multiple versions in the future, and it is easier for programmers to remember one reasonable name for tasks that are functionally identical except for parameter types.

Understanding Overload Resolution

When a method call *could* execute multiple overloaded method versions, C# determines which method to execute using a process called **overload resolution**. For example, suppose that you create a method with the following declaration:

```
private static void MyMethod(double d)
```

You can call this method using a `double` argument, as in the following:

```
MyMethod(2.5);
```

You can also call this method using an `int` argument, as in the following:

```
MyMethod(4);
```

The call that uses the `int` argument works because an `int` can automatically be promoted to a `double`. In Chapter 2 you learned that when an `int` is promoted to a `double`, the process is called an *implicit conversion* or *implicit cast*.

Suppose that you create overloaded methods with the following declarations:

```
private static void MyMethod(double d)
private static void MyMethod(int i)
```

If you then call `MyMethod()` using an integer argument, both methods are **applicable methods**. That means either method on its own could accept a call that uses an `int`. However, if both methods exist in the same class (making them overloaded), the second version will execute because it is a better match for the method call. The rules that determine which method version to call are known as **betterness rules**.

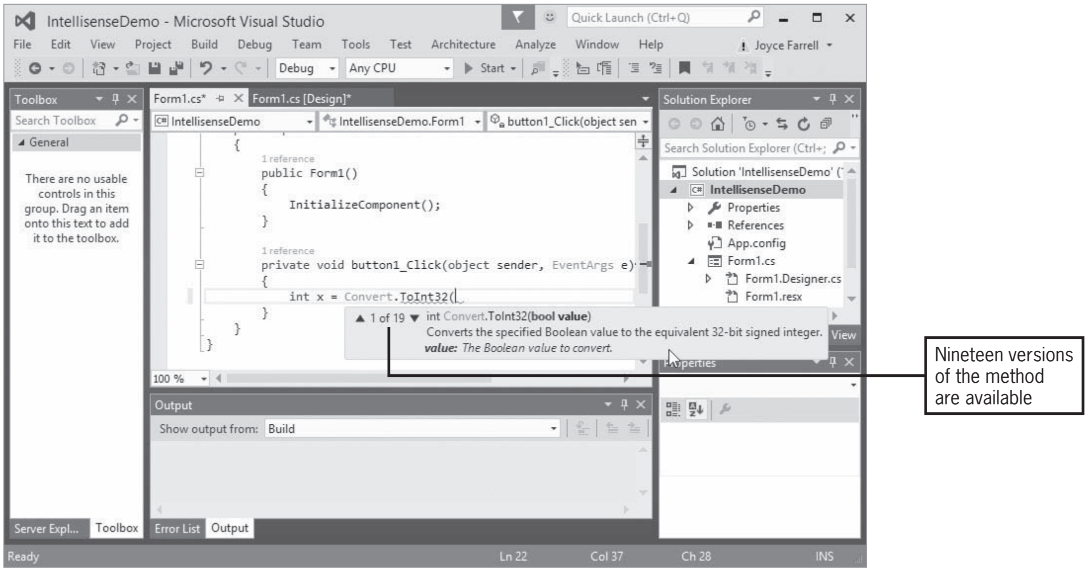
Betterness rules are similar to the implicit conversion rules you learned in Chapter 2. For example, although an `int` could be accepted by a method that accepts an `int`, a `float`, or a `double`, an `int` is the best match. If no method version with an `int` parameter exists, then a `float` is a better match than a `double`. Table 8-1 shows the betterness rules for several data types.

Data Type	Conversions Are Better in This Order
`byte`	`short, ushort, int, uint, long, ulong, float, double, decimal`
`sbyte`	`short, int, long, float, double, decimal`
`short`	`int, long, float, double, decimal`
`ushort`	`int, uint, long, ulong, float, double, decimal`
`int`	`long, float, double, decimal`
`uint`	`long, ulong, float, double, decimal`
`long`	`float, double, decimal`
`ulong`	`float, double, decimal`
`float`	`double`
`char`	`ushort, int, uint, long, ulong, float, double, decimal`

Table 8-1 Betterness rules for data type conversion

Discovering Built-In Overloaded Methods

When you use the IDE to create programs, Visual Studio's IntelliSense features provide information about methods you are using. For example, Figure 8-19 shows the IDE just after the programmer has typed the opening parenthesis to the `Convert.ToInt32()` method. Notice the drop-down list that indicates `Convert.ToInt32(bool value)` is just 1 of 19 overloaded versions of the method. You can click the nodes to scroll through the rest of the list and see that other versions accept `bool`, `byte`, `char`, and so on. When retrieving an entry from a `TextBox` on a `Form`, your intention is to use the `Convert.ToInt32(string value)` version of the method, but many overloaded versions are available for your convenience. C# could have included multiple methods with different names such as `ConvertBool.ToInt32()` and `ConvertString.ToInt32()`, but having a single method with the name `Convert.ToInt32()` that takes many different argument types makes it easier for you to remember the method name and to use it. As you work with the IDE, you will examine many such methods.



```
    {
        1 reference
        public Form1()
        {
            InitializeComponent();
        }

        1 reference
        private void button1_Click(object sender, EventArgs e)
        {
            int x = Convert.ToInt32(
        }
    }
}
```

▲ 1 of 19 ▼ int Convert.ToInt32(**bool value**)
Converts the specified Boolean value to the equivalent 32-bit signed integer.
value: *The Boolean value to convert.*

IntellisenseDemo - Microsoft Visual Studio
Nineteen versions of the method are available

Figure 8-19 Examining the overloaded versions of `Convert.ToInt32()`

TWO TRUTHS & A LIE

Overloading Methods

1. The following methods are overloaded:

```
private static void MethodA(int a)
private static void MethodA(double b)
```

2. The following methods are overloaded:

```
private static void MethodC(int c)
private static void MethodD(int c)
```

3. The following methods are overloaded:

```
private static void MethodE(int e)
private static void MethodE(int e, int f)
```

The false answer is #2. Overloaded methods must have the same name but different parameter lists.

You Do It

Overloading Methods

In the next steps, you overload methods that correctly triple a parameter that might be an integer or a string.

1. Start a new program named **OverloadedTriples**, and create a method that triples and displays an integer parameter as follows:

```
private static void Triple(int num)
{
   const int MULT_FACTOR = 3;
   WriteLine("{0} times {1} is {2}\n",
      num, MULT_FACTOR, num * MULT_FACTOR);
}
```

2. Create a second method with the same name that takes a string parameter. Assume you want to define tripling a message as displaying it three times, separated by tabs.

```
private static void Triple(string message)
{
   WriteLine("{0}\t{0}\t{0}\n", message);
}
```

(continues)

(continued)

3. Position your cursor at the top of the file, and add a `using` statement, class header, and opening curly brace so the overloaded `Triple()` methods will be contained in a class named `OverloadedTriples`.

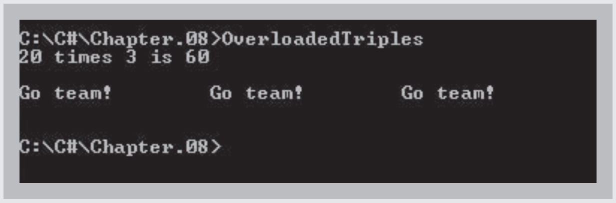
```
using static System.Console;
class OverloadedTriples
{
```

4. Position your cursor at the bottom of the file, and add the closing curly brace for the `OverloadedTriples` class.

5. Position your cursor after the opening curly brace for the class. On a new line, insert a `Main()` method that declares an integer and a string and, in turn, passes each to the appropriate `Triple()` method. Notice that as you type the parenthesis after the `Triple()` method, the IDE displays your two overloaded `Triple()` methods.

```
static void Main()
{
   int num = 20;
   string message = "Go team!";
   Triple(num);
   Triple(message);
}
```

6. Save the file, and then compile and execute the program. Figure 8-20 shows the output. Even though the same method name is used in the two method calls, the appropriate overloaded method executes each time.



```
C:\C#\Chapter.08>OverloadedTriples
20 times 3 is 60

Go team!        Go team!        Go team!


C:\C#\Chapter.08>
```

Figure 8-20 Output of the `OverloadedTriples` program

Avoiding Ambiguous Methods

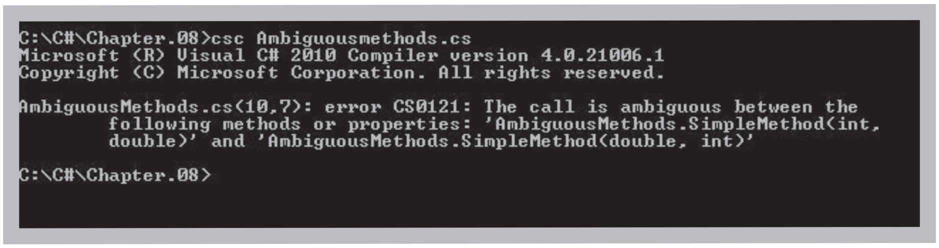
When you overload a method, you run the risk of creating **ambiguous** methods—a situation in which the compiler cannot determine which method to use. Every time you call a method, the compiler decides whether a suitable method exists; if so, the method executes, and if not, you receive an error message.

For example, suppose that you write two versions of a simple method, as in the program in Figure 8-21. The class contains two versions of a method named `SimpleMethod()`—one that takes a `double` and an `int`, and one that takes an `int` and a `double`.

```
using static System.Console;
class AmbiguousMethods
{
   static void Main()
   {
      int iNum = 20;
      double dNum = 4.5;
      SimpleMethod(iNum, dNum); // calls first version
      SimpleMethod(dNum, iNum); // calls second version
      SimpleMethod(iNum, iNum); // error! Call is ambiguous.
   }
   private static void SimpleMethod(int i, double d)
   {
      WriteLine("Method receives int and double");
   }
   private static void SimpleMethod(double d, int i)
   {
      WriteLine("Method receives double and int");
   }
}
```

Figure 8-21 Program containing an ambiguous method call

In the `Main()` method in Figure 8-21, a call to `SimpleMethod()` with an integer argument first and a `double` argument second executes the first version of the method, and a call to `SimpleMethod()` with a `double` argument first and an integer argument second executes the second version of the method. With each of these calls, the compiler can find an exact match for the arguments you send. However, if you call `SimpleMethod()` using two integer arguments, as in the shaded statement, an ambiguous situation arises because there is no exact match for the method call. Because the first integer could be promoted to a `double` (matching the second version of the overloaded method), or the second integer could be promoted to a `double` (matching the first version), the compiler does not know which version of `SimpleMethod()` to use, and the program will not compile or execute. Figure 8-22 shows the error message that is generated.



```
C:\C#\Chapter.08>csc Ambiguousmethods.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

AmbiguousMethods.cs(10,7): error CS0121: The call is ambiguous between the
        following methods or properties: 'AmbiguousMethods.SimpleMethod(int,
        double)' and 'AmbiguousMethods.SimpleMethod(double, int)'

C:\C#\Chapter.08>
```

Figure 8-22 Error message generated by an ambiguous method call

An overloaded method is not ambiguous on its own—it becomes ambiguous only if you create an ambiguous situation. A program with potentially ambiguous methods will run without problems if you make no ambiguous method calls. For example, if you remove the shaded statement from Figure 8-21 that calls `SimpleMethod()` using two integer arguments, the program compiles and executes.

If you remove one of the versions of `SimpleMethod()` from the program in Figure 8-21, then the method call that uses two integer arguments would work, because one of the integers could be promoted to a `double`. However, then one of the other method calls would fail.

Methods can be overloaded correctly by providing different parameter lists for methods with the same name. Methods with identical names that have identical parameter lists but different return types are not overloaded—they are illegal. For example, the following two methods cannot coexist within a class:

```
private static int AMethod(int x)
private static void AMethod(int x)
```

The compiler determines which of several versions of a method to call based on parameter lists. When the method call `AMethod(17);` is made, the compiler will not know which method to execute because both possibilities take an integer argument. Similarly, the following method could not coexist with either of the previous versions:

```
private static void AMethod(int someNumber)
```

Even though this method uses a different local identifier for the parameter, its parameter list—a single integer—is still the same to the compiler.

Watch the video *Overloading Methods*.

TWO TRUTHS & A LIE

Avoiding Ambiguous Methods

1. The following methods are potentially ambiguous:

```
private static int Method1(int g)
private static int Method1(int g, int h)
```

2. The following methods are potentially ambiguous:

```
private static double Method2(int j)
private static void Method2(int k)
```

3. The following methods are potentially ambiguous:

```
private static void Method3(string m)
private static string Method3(string n)
```

The false answer is #1. Those methods are not ambiguous because they have different parameter lists. Their matching return types do not cause ambiguity.

Using Optional Parameters

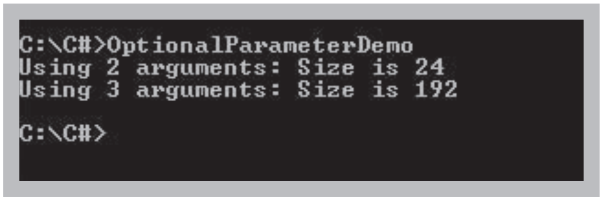
Sometimes it is useful to create a method that allows one or more arguments to be omitted from the method call. An optional parameter is not required; if you don't send a value as an argument, a default value is automatically supplied. You make a parameter optional by providing a value for it in the method declaration. Only value parameters can be given default values; those that use `ref`, `out`, and `params` cannot have default values. Any optional parameters in a parameter list must appear to the right of the mandatory parameters.

For example, suppose that you write a method that calculates either the area of a square or the volume of a cube, depending on whether two or three arguments are sent to it. Figure 8-23 shows such a method; notice the shaded default value provided for the third parameter. When you want a cube's volume, you can pass three arguments to the method: length, width, and height. When you want a square's area, you pass only two parameters, and a default value of 1 is used for the height. In this example, the height is a default parameter and is optional; the length and width, which are not provided with default values, are mandatory parameters. That is, at least two arguments must be sent to the

`DisplaySize()` method, but three *can* be sent. In the program in Figure 8-23, calling `DisplaySize(4, 6)` has the same effect as calling `DisplaySize(4, 6, 1)`. Figure 8-24 shows the execution of the OptionalParameterDemo program.

```
using static System.Console;
class OptionalParameterDemo
{
   static void Main()
   {
      Write("Using 2 arguments: ");
      DisplaySize(4, 6);
      Write("Using 3 arguments: ");
      DisplaySize(4, 6, 8);
   }
   private static void DisplaySize(int length, int width, int height = 1)
   {
      int area = length * width * height;
      WriteLine("Size is {0}", area);
   }
}
```

Figure 8-23 The OptionalParameterDemo class



```
C:\C#>OptionalParameterDemo
Using 2 arguments: Size is 24
Using 3 arguments: Size is 192

C:\C#>
```

Figure 8-24 Execution of the OptionalParameterDemo program

If you assign a default value to any variable in a method's parameter list, then all parameters to the right of that parameter must also have default values. Table 8-2 shows some examples of valid and invalid method declarations.

Method Declaration	Explanation
`private static void M1(int a, int b, int c, int d = 10)`	Valid. The first three parameters are mandatory and the last one is optional.
`private static void M2(int a, int b = 3, int c)`	Invalid. Because b has a default value, c must also have one.
`private static void M3(int a = 3, int b = 4, int c = 5)`	Valid. All parameters are optional.
`private static void M4(int a, int b, int c)`	Valid. All parameters are mandatory.
`private static void M5(int a = 4, int b, int c = 8)`	Invalid. Because a has a default value, both b and c must have default values.

Table 8-2 Examples of valid and invalid optional parameter method declarations

When you call a method that contains default parameters, you can always choose to provide a value for every parameter. However, if you omit an argument when you call a method that has default parameters, then you must do one of the following:

- You must leave out all unnamed arguments to the right of the last argument you use.
- You can name arguments.

Leaving Out Unnamed Arguments

When calling a method with optional parameters (and using unnamed arguments), you must leave out any arguments to the right of the last one used. In other words, once an argument is left out, you must leave out all the arguments that would otherwise follow.

For example, assume you have declared a method as follows:

```
private static void Method1(int a, char b, int c = 22, double d = 33.2)
```

Table 8-3 shows some legal and illegal calls to this method.

Call to `Method1()`	Explanation
`Method1(1, 'A', 3, 4.4);`	Valid. The four arguments are assigned to the four parameters.
`Method1(1, 'K', 9);`	Valid. The three arguments are assigned to `a`, `b`, and `c` in the method, and the default value of 33.2 is used for d.
`Method1(5, 'D');`	Valid. The two arguments are assigned to a and b in the method, and the default values of 22 and 33.2 are used for `c` and `d`, respectively.
`Method1(1);`	Invalid. `Method1()` requires at least two arguments for the first two parameters.
`Method1();`	Invalid. `Method1()` requires at least two arguments for the first two parameters.
`Method1(3, 18.5);`	Invalid. The first argument, 3, can be assigned to `a`, but the second argument must be `type char`.
`Method1(4, 'R', 55.5);`	Invalid. The first argument, 4, can be assigned to `a`, and the second argument, 'R', can be assigned to `b`, but the third argument must be type `int`. When arguments are unnamed, you cannot "skip" parameter `c`, use its default value, and assign 55.5 to parameter `d`.

Table 8-3 Examples of legal and illegal calls to `Method1()`

Using Named Arguments

You can leave out optional arguments in a method call if you pass the remaining arguments by name. A **named argument** is a method argument that is preceded with the name of the called method's parameter to which it will be assigned. Named arguments can appear in any order, but they must appear after all the unnamed arguments have been listed. Each unnamed argument is also known as a **positional argument**. You name an argument using its parameter name and a colon before the value.

In Chapter 7, you learned that if you use the IntelliSense feature of Visual Studio, you can discover a method's parameter names. See Appendix C for more details.

For example, assume you have declared a method as follows:

```
private static void Method2(int a, char b, int c = 22, double d = 33.2)
```

Table 8-4 shows some legal and illegal calls to this method.

Call to `Method2()`	Explanation
`Method2(1, 'A');`	Valid. The two arguments are assigned to `a` and `b` in the method, and the default values of 22 and 33.2 are used for `c` and `d`, respectively.
`Method2(2, 'E', 3);`	Valid. The three arguments are assigned to `a`, `b`, and `c`. The default value 33.2 is used for `d`.
`Method2(2, 'E', c : 3);`	Valid. This call is identical to the one above.
`Method2(1, 'K', d : 88.8);`	Valid. The first two arguments are assigned to `a` and `b`. The default value 22 is used for `c`. The named value 88.8 is used for `d`.
`Method2(d : 2.1, b : 'A', c : 88, a: 12);`	Valid. All the arguments are assigned to parameters whether they are listed in order or not.
`Method2(5, 'S', d : 7.4, c: 9);`	Valid. The first two arguments are assigned to `a` and `b`. Even though the values for `c` and `d` are not listed in order, they are assigned correctly.
`Method2(d : 11.1, 6, 'P');`	Invalid. This call contains an unnamed argument after the first named argument. Named arguments must appear after all unnamed arguments.

Table 8-4 Examples of legal and illegal calls to `Method2()`

Advantages to Using Named Arguments

Optional parameters can provide a convenient shortcut to writing multiple overloaded versions of a method. For example, suppose that you want to create a method that adds your signature to business letters. Usually, you want the signature to be *Sincerely, James O'Hara*. However, occasionally you want to change the name to a more casual *Jim*. You could write two method versions, as shown on the top in Figure 8-25, or you could write one version, as shown on the bottom. With both versions, you can call `Closing()` with or without an argument. The version with the optional parameter is less work when you write the method. It is also less work if you later want to modify the method—for example, to change *Sincerely* to *Best wishes* or some other closing.

```
// Overloaded implementations of Closing()
private static void Closing()
{
   WriteLine("Sincerely,");
   WriteLine("James O'Hara");
}
private static void Closing(string name)
{
   WriteLine("Sincerely,");
   WriteLine(name);
}

// Single implementation of Closing() with optional parameter
private static void Closing(string name = "James O'Hara")
{
   WriteLine("Sincerely,");
   WriteLine(name);
}
```

Figure 8-25 Two ways to implement `Closing()` to accept a name parameter or not

Another advantage to using named parameters is to make your programs more self-documenting. Programs that are **self-documenting** provide built-in explanations that make the code clearer to readers and therefore easier for others to modify in the future. For example, suppose that you encounter the following method call:

```
DisplayEmployeeData(empNumber, true);
```

From the method call, it is not clear what `true` means. Perhaps it means that only certain parts of the record should be displayed, or that the employee is a current employee, or that the employee's salary should be hidden from view. The following method call makes your intentions clearer:

```
DisplayEmployeeData(empNumber, shouldHideSalary : true);
```

Of course, a program comment adjacent to the statement could also provide clarity.

Disadvantages to Using Named Arguments

A major disadvantage to using named arguments is that the calling method becomes linked to details within the method. When the calling method must know parameter names within the called method, an important principle of programming—method implementation hiding—is compromised. If the parameter name in the called method is changed in the future, the client method also will have to change.

Suppose that a payroll program contains two methods used in paycheck calculations. The first method, shown in Figure 8-26, accepts three parameters—`hours`, `rate`, and `bonus`. A gross pay value is calculated by multiplying `hours` by `rate`, and a bonus value is assigned based on the number of hours worked. The second method sums its `gross` and `bonus` parameters. The intention is that an employee who works 40 hours at $10 per hour receives $400 gross pay, plus a $100 bonus, for a total of $500.

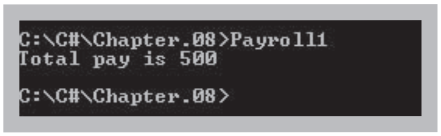
```
private static double ComputeGross(double hours, double rate,
  out double bonus)
{
   double gross = hours * rate;
   if (hours >= 40)
      bonus = 100;
   else
      bonus = 50;
   return gross;
}
private static double ComputeTotalPay(double gross, double bonus)
{
   double total = gross + bonus;
   return total;
}
```

Figure 8-26 Two payroll program methods

Figure 8-27 shows the `Main()` method of a program that assigns 40 to `hours` and 10.00 to `rate`. Then, the method calls the `ComputeTotalPay()` method by passing it the result of `ComputeGross()` and the value of `bonus` that is set within `ComputeGross()`. Figure 8-28 shows that the employee's total pay is correctly calculated as $500.

```
static void Main()
{
   double hours = 40;
   double rate = 10.00;
   double bonus = 0;
   double totalPay;
   totalPay = ComputeTotalPay(ComputeGross(hours, rate, out bonus),
     bonus);
   WriteLine("Total pay is {0}", totalPay);
}
```

Figure 8-27 A `Main()` method that calls `ComputeTotalPay()` using positional arguments



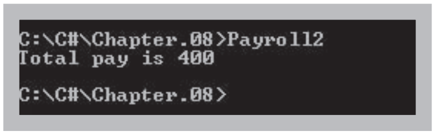
```
C:\C#\Chapter.08>Payroll1
Total pay is 500

C:\C#\Chapter.08>
```

Figure 8-28 Execution of the `Main()` method in Figure 8-27

A possible disadvantage to using named arguments occurs when a named parameter value is an expression instead of a constant. Figure 8-29 shows a `Main()` method that calls `ComputeTotalPay()` using named arguments. The only difference from the program in Figure 8-27 is the shaded method call that uses named arguments. In this case, when `ComputeTotalPay()` is called, first 0 is sent to `bonus`, and then `ComputeGross()` is called to assign a value to `gross`. However, the bonus alteration in `ComputeGross()` is too late because `bonus` has already been assigned. The program output is incorrect, as shown in Figure 8-30.

```
static void Main()
{
   double hours = 40;
   double rate = 10.00;
   double bonus = 0;
   double totalPay;
   totalPay = ComputeTotalPay(bonus: bonus,
      gross: ComputeGross(hours, rate, out bonus));
   WriteLine("Total pay is {0}", totalPay);
}
```

Figure 8-29 A `Main()` method that calls `ComputeTotalPay()` using named arguments



```
C:\C#\Chapter.08>Payroll2
Total pay is 400

C:\C#\Chapter.08>
```

Figure 8-30 Execution of the `Main()` method in Figure 8-29

In Chapter 4, you learned that when you combine operands in a Boolean expression using && or `||`, you risk side effects because of short-circuit evaluation. The situation with named arguments is similar. When a named argument is an expression, there can be unintended consequences.

Overload Resolution with Named and Optional Arguments

Named and optional arguments affect overload resolution. The rules for betterness on argument conversions are applied only for arguments that are given explicitly; in other words, omitted optional arguments are ignored for betterness purposes. For example, suppose that you have the following methods:

```
private static void AMethod(int a, double b = 2.2)
private static void AMethod(int a, char b = 'H')
```

Both are applicable methods when a call is made using an integer argument; in other words, if either method existed alone, the call `AMethod(12)` would work. When the two methods coexist, however, neither is "better." Because C# cannot determine which one is better, the code will not compile.

If two signatures are equally good, the one that does not omit optional parameters is considered better. For example, suppose that you have two methods as follows:

```
private static void BMethod(int a)
private static void BMethod(int a, char b = 'B')
```

If either method existed alone, the call `BMethod(12)` would work, but when the two coexist, the first version is better because no optional parameters are omitted in the call.

Watch the video *Using Optional Method Parameters*.

TWO TRUTHS & A LIE

Using Optional Parameters

1. An optional parameter to a method is one for which an argument can be a value parameter, a `ref` parameter, or an `out` parameter.
2. You make a parameter optional by providing a value for it in the method declaration.
3. If you assign a default value to any variable in a method's parameter list, then all parameters to the right of that parameter must also have default values.

The false statement is #1. An optional parameter to a method is one for which a default value is automatically supplied if you do not explicitly send one as an argument. Only value parameters can be optional.

Chapter Summary

- Method parameters can be mandatory or optional. Mandatory parameters include value parameters without default values, reference parameters, output parameters, and parameter arrays. A value parameter can be optional if it is supplied with a default value.
- When you overload a C# method, you write multiple methods with a shared name but different parameter lists. The compiler understands your meaning based on the combination of arguments you use with the method. When a method call could execute multiple overloaded method versions, C# determines which method to execute using a process called *overload resolution*.
- When you overload a method, you run the risk of creating an ambiguous situation—one in which the compiler cannot determine which method to use. Methods can be overloaded correctly by providing different parameter lists for methods with the same name.
- An optional parameter to a method is a value parameter for which a default value is automatically supplied if you do not explicitly send one as an argument. You make a parameter optional by providing a value for it in the method declaration. When calling a method with optional parameters, you must leave out all unnamed arguments to the right of the last one used, or you can name arguments.

Key Terms

Mandatory parameters are method parameters for which an argument is required in every method call.

An **optional parameter** to a method is one for which a default value is automatically supplied if you do not explicitly send one as an argument; a parameter is optional when it is given a value in the method declaration.

A **value parameter** in a method header receives a copy of the value passed to it.

A **reference parameter** in a method header receives the parameter's address; a variable sent to a reference parameter is required to have been assigned a value before it is used in the method call.

An **output parameter** in a method header receives the parameter's address; a variable sent to an output parameter is not required to have been assigned a value before it is used in the method call.

A **parameter array** is a local array declared within a method header that can accept any number of elements of the same data type.

Overloading involves the ability to write multiple versions of a method using the same method name but different parameter lists.

A method's **signature** is composed of its name and parameter list.

Overload resolution is the process of determining which of multiple applicable methods is the best match for a method call.

Applicable methods are all the methods that could be used by a method call.

Betterness rules are the rules that determine the best overloaded method to execute based on the arguments in a method call.

Ambiguous methods are overloaded methods from which the compiler cannot determine which one to use.

A **named argument** is a method argument that is preceded with the name of the called method's parameter to which it will be assigned.

A **positional argument** is an unnamed method argument that is assigned to a parameter list based on its position in the method call.

Self-documenting describes programs that provide built-in explanations to make the code clearer to readers and therefore easier for others to modify in the future.

Review Questions

1. A mandatory parameter ____________________.
   - a. is any argument sent to a method
   - b. is preceded by the keyword `man`
   - c. requires an argument to be sent from a method call
   - d. All of the above are true.

2. Which is *not* a type of method parameter in C#?
   - a. value
   - b. reference
   - c. forensic
   - d. output

3. Which type of method parameter receives the address of the variable passed in?
   - a. a value parameter
   - b. a reference parameter
   - c. an output parameter
   - d. two of the above

4. When you declare a value parameter, you precede its name with ____________________.
   - a. nothing
   - b. a data type
   - c. the keyword `val` and a data type
   - d. the keyword `ref` and its data type

5. Assume that you declare a variable as `int x = 100;` and correctly pass it to a method with the declaration `private static void IncreaseValue(ref int x)`. There is a single statement within the `IncreaseValue()` method: `x = x + 25;`. Back in the `Main()` method, after the method call, what is the value of `x`?
   a. 100
   b. 125
   c. It is impossible to tell.
   d. The program will not run.

6. Assume that you declare a variable as `int x = 100;` and correctly pass it to a method with the declaration `private static void IncreaseValue(int x)`. There is a single statement within the `IncreaseValue()` method: `x = x + 25;`. Back in the `Main()` method, after the method call, what is the value of `x`?
   a. 100
   b. 125
   c. It is impossible to tell.
   d. The program will not run.

7. Which of the following is a difference between a reference parameter and an output parameter?
   a. A reference parameter receives a memory address; an output parameter does not.
   b. A reference parameter occupies a unique memory address; an output parameter does not.
   c. A reference parameter requires an initial value; an output parameter does not.
   d. A reference parameter does not require an initial value; an output parameter does.

8. A parameter array ____________________.
   a. is declared using the keyword `params`
   b. can accept any number of arguments of the same data type
   c. Both of these are true.
   d. Neither of these is true.

9. Assume that you have declared a method with the following header: `private static void DisplayScores(params int[] scores)` Which of the following method calls is valid?
   a. `DisplayScores(20);`
   b. `DisplayScores(20, 33);`
   c. `DisplayScores(20, 30, 90);`
   d. All of the above are valid.

10. Correctly overloaded methods must have the same ____________________.
    a. return type
    b. identifier
    c. parameter list
    d. All of the above.

11. Methods are ambiguous when they ____________________.
    a. are overloaded
    b. are written in a confusing manner
    c. are indistinguishable to the compiler
    d. have the same parameter type as their return type

12. Which of the following pairs of method declarations represent correctly overloaded methods?
    a. `private static void MethodA(int a)`
       `private static void MethodA(int b, double c)`
    b. `private static void MethodB(double d)`
       `private static void MethodB()`
    c. `private static double MethodC(int e)`
       `private static double MethodD(int f)`
    d. Two of these are correctly overloaded methods.

13. Which of the following pairs of method declarations represent correctly overloaded methods?
    a. `private static void Method(int a)`
       `private static void Method(int b)`
    b. `private static void Method(double d)`
       `private static int Method()`
    c. `private static double Method(int e)`
       `private static int Method(int f)`
    d. Two of these are correctly overloaded methods.

14. The process of determining which overloaded version of a method to execute is overload ____________________.
    a. confusion
    b. infusion
    c. revolution
    d. resolution

15. When one of a method's parameters is optional, it means that ____________________.
    a. no arguments are required in a call to the method
    b. a default value will be assigned to the parameter if no argument is sent for it
    c. a default value will override any argument value sent to it
    d. you are not required to use the parameter within the method body

16. Which of the following is an illegal method declaration?

    a. `private static void CreateStatement(int acctNum, double balance = 0.0)`

    b. `private static void CreateStatement(int acctNum = 0, double balance)`

    c. `private static void CreateStatement(int acctNum = 0, double balance = 0)`

    d. All of these are legal.

17. Assume you have declared a method as follows:
    `private static double ComputeBill(int acct, double price, double discount = 0)`
    Which of the following is a legal method call?

    a. `ComputeBill();`

    b. `ComputeBill(1001);`

    c. `ComputeBill(1001, 200.00);`

    d. None of the above is legal.

18. Assume you have declared a method as follows:
    `private static double CalculateDiscount(int acct = 0, double price = 0, double discount = 0)`
    Which of the following is a legal method call?

    a. `CalculateDiscount();`

    b. `CalculateDiscount(200.00);`

    c. `CalculateDiscount(3000.00, 0.02);`

    d. None of the above is legal.

19. Assume you have declared a method as follows:
    `private static double DisplayData(string name = "XX", double amount = 10.0)`
    Which of the following is an illegal method call?

    a. `DisplayData(name : "Albert");`

    b. `DisplayData(amount : 200, name : "Albert");`

    c. `DisplayData(amount : 900.00);`

    d. All of these are legal.

20. Suppose that you have declared an integer array named `scores`, and you make the following method call:
    `TotalScores(scores, num : 1);`
    Of the following overloaded method definitions, which would execute?
    a. `private static void TotalScores(int[] scores)`
    b. `private static void TotalScores(int[] scores, int num)`
    c. `private static void TotalScores(int[] scores, int num = 10, int code = 10)`
    d. The program would not compile.

Exercises

Programming Exercises

1. Unless otherwise indicated, for each of the following programming exercises, you may choose to write a console-based or GUI application, or both.
    a. Create a program named **Reverse3** whose `Main()` method declares three integers named `firstInt`, `middleInt`, and `lastInt`. Assign values to the variables, display them, and then pass them to a method that accepts them as reference variables, places the first value in the `lastInt` variable, and places the last value in the `firstInt` variable. In the `Main()` method, display the three variables again, demonstrating that their positions have been reversed.
    b. Create a new program named **Reverse4**, which contains a method that reverses the positions of four variables. Write a `Main()` method that demonstrates the method works correctly.
2. Create a program named **IntegerFacts** whose `Main()` method declares an array of 20 integers. Call a method to interactively fill the array with any number of values up to 20 or until a sentinel value is entered. If an entry is not an integer, reprompt the user. Call a second method that accepts `out` parameters for the highest value in the array, lowest value in the array, sum of the values in the array, and arithmetic average. In the `Main()` method, display all the statistics.
3. Create a program for The Cactus Cantina named **FoodOrder** that accepts a user's choice from the options in the accompanying table. Allow the user to enter either an integer item number or a string description. Pass the user's entry to one of two overloaded `GetDetails()` methods, and then display a returned string with all the

order details. The method version that accepts an integer looks up the description and price; the version that accepts a string description looks up the item number and price. The methods return an appropriate message if the item is not found.

Item number	Description	Price
20	Enchilada	2.95
23	Burrito	1.95
25	Taco	2.25
31	Tostada	3.10

4. Create a program named **Auction** that allows a user to enter an amount bid on an online auction item. Include three overloaded methods that accept an `int`, `double`, or `string` bid. Each method should display the bid and indicate whether it is over the minimum acceptable bid of $10. If the bid is a `string`, accept it only if one of the following is true: it is numeric and preceded with a dollar sign, or it is numeric and followed by the word *dollars*. Otherwise, display a message that indicates the format was incorrect.

5. Create a program named **BonusCalculation** that includes two overloaded methods—one that accepts a salary and a bonus expressed as `double`s (for example, 600.00 and 0.10, where 0.10 represents a 10 percent bonus), and one that accepts a salary as a `double` and a bonus an integer (for example, 600.00 and 50, where 50 represents a $50 bonus). Each method displays the salary, the bonus in dollars, and the total. Include a `Main()` method that demonstrates each method.

6. Write a program named **InputMethodDemo2** that eliminates the repetitive code in the `InputMethod()` in the `InputMethodDemo` program in Figure 8-5. Rewrite the program so the `InputMethod()` contains only two statements:

```
one = DataEntry("first");
two = DataEntry("second");
```

7. Write a program named **Averages** that includes a method that accepts any number of numeric parameters, displays them, and displays their average. Demonstrate that the method works correctly when passed one, two, or three numbers, or an array of numbers.

8. Write a program named **Desks2** that modifies the `Desks` program you created in Chapter 7 so that all the methods are `void` methods and receive `ref` or `out` parameters as appropriate.

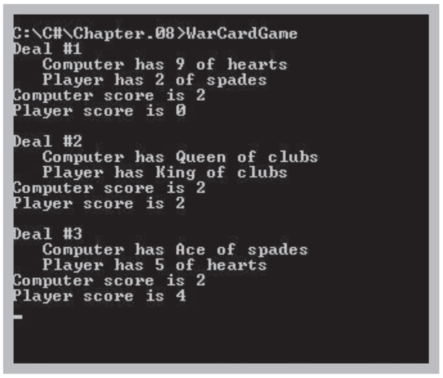
9. Write a program named **SortWords** that includes a method that accepts any number of words and sorts them in alphabetical order. Demonstrate that the program works correctly when the method is called with one, two, five, and ten words.

10. Write a program named **Movie** that contains a method that accepts and displays two parameters: a string name of a movie and an integer running time in minutes. Provide a default value for the minutes so that if you call the method without an integer argument, `minutes` is set to 90. Write a `Main()` method that proves you can call the movie method with only a `string` argument as well as with a `string` and an integer.

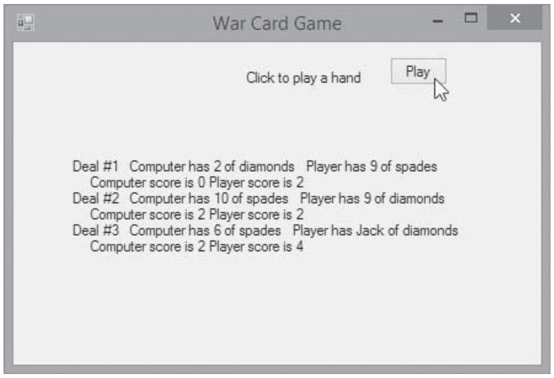
11. In the card game War, a deck of playing cards is divided between two players. Each player exposes a card; the player whose card has the higher value wins possession of both exposed cards. Create a console-based computerized game of War named **WarCardGame** in which a standard 52-card deck is randomly divided between two players, one of which is the computer. Reveal one card for the computer and one card for the player at a time. Award two points for the player whose card has the higher value. (For this game the king is the highest card, followed by the queen and jack, then the numbers 10 down to 2, and finally the ace.) If the computer and player expose cards with equal values in the same turn, award one point to each. At the end of the game, all 52 cards should have been played only once, and the sum of the player's and computer's score will be 52.

    a. Use an array of 52 integers to store unique values for each card. Write a method named `FillDeck()` that places 52 unique values into this array. Write another method named `SelectCard()` that you call twice on each deal to select a unique card for each player, with no repetition of cards in 26 deals. (To pause the play between each dealt hand, use a call to `ReadLine()`.)

       The left side of Figure 8-31 shows the start of a typical program execution. (*Caution*: This is a difficult exercise!)



```
C:\C#\Chapter.08>WarCardGame
Deal #1
   Computer has 9 of hearts
   Player has 2 of spades
Computer score is 2
Player score is 0

Deal #2
   Computer has Queen of clubs
   Player has King of clubs
Computer score is 2
Player score is 2

Deal #3
   Computer has Ace of spades
   Player has 5 of hearts
Computer score is 2
Player score is 4
```



```
War Card Game

Click to play a hand   Play

Deal #1   Computer has 2 of diamonds   Player has 9 of spades
   Computer score is 0 Player score is 2
Deal #2   Computer has 10 of spades   Player has 9 of diamonds
   Computer score is 2 Player score is 2
Deal #3   Computer has 6 of spades   Player has Jack of diamonds
   Computer score is 2 Player score is 4
```

Figure 8-31 Start of typical execution of console-based and GUI WarCardGame programs

b. Create a GUI version of the War card game described in Exercise 11a, and name it **WarCardGameGUI**. Let the user click a button to deal the cards, then make that button invisible and expose a Play button. Each time the user clicks Play, a pair of cards is revealed. To keep the `Frame` size reasonable, you might want to erase the output label's contents every four hands or so. The right side of Figure 8-31 shows a typical game in progress.

Debugging Exercises

1. Each of the following files in the Chapter.08 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, DebugEight1.cs will become FixedDebugEight1.cs.
   a. DebugEight1.cs
   b. DebugEight2.cs
   c. DebugEight3.cs
   d. DebugEight4.cs

Case Problems

1. In Chapter 7, you modified the **GreenvilleRevenue** program to include a number of methods. Now modify every data entry statement to use a `TryParse()` method to ensure that each piece of data is the correct type. Any invalid user entries should generate an appropriate message, and the user should be required to reenter the data.
2. In Chapter 7, you modified the **MarshallsRevenue** program to include a number of methods. Now modify every data entry statement to use a `TryParse()` method to ensure that each piece of data is the correct type. Any invalid user entries should generate an appropriate message, and the user should be required to reenter the data.