msdn training

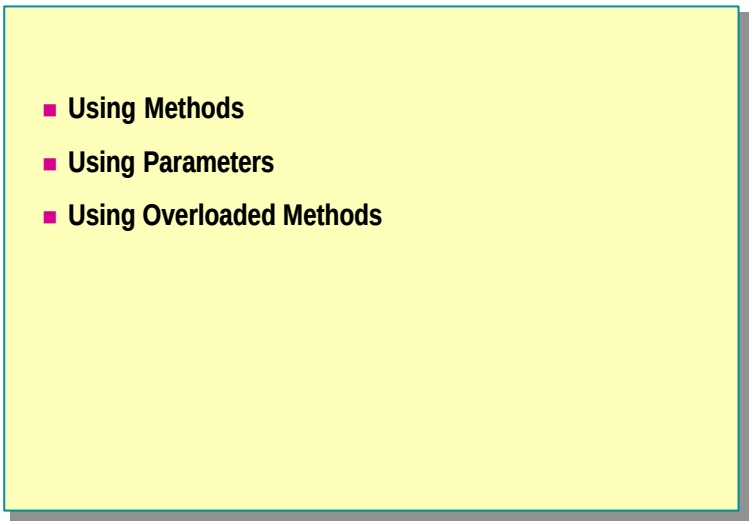
Module 5: Methods and Parameters

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft

Overview



In designing most applications, you divide the application into functional units. This is a central principle of application design because small sections of code are easier to understand, design, develop, and debug. Dividing the application into functional units also allows you to reuse functional components throughout the application.

In C#, you structure your application into classes that contain named blocks of code; these are called methods. A *method* is a member of a class that performs an action or computes a value.

After completing this module, you will be able to:

- Create static methods that accept parameters and return values.
- Pass parameters to methods in different ways.
- Declare and use overloaded methods.

Using Methods

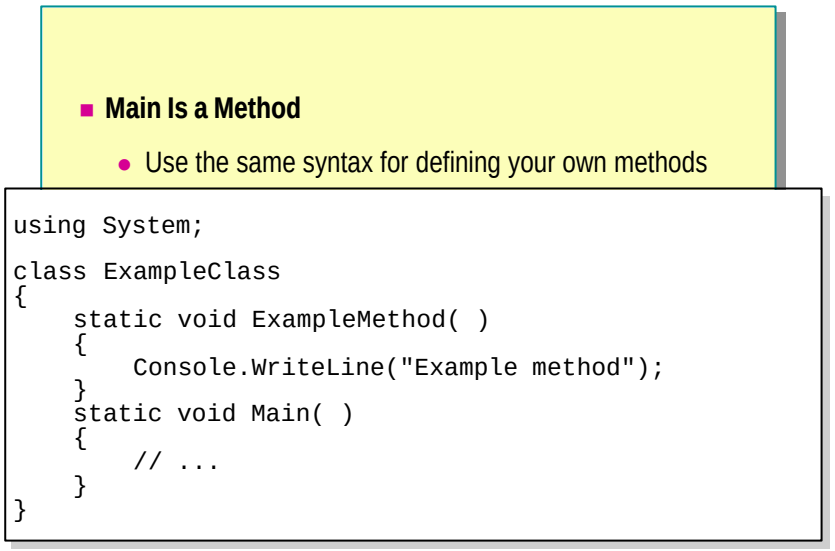
- Defining Methods
- Calling Methods
- Using the return Statement
- Using Local Variables
- Returning Values

In this section, you will learn how to use methods in C#. Methods are important mechanisms for struc turing program code. You will learn how to create methods and how to call them from within a single class and from one class to another.

You will learn how to use local variables, as well as how to allocate and destroy them.

You will also learn how to return a value from a method, and how to use parameters to transfer data into and out of a method.

Defining Methods



A method is group of C# statements that have been brought together and given a name. Most modern programming languages have a similar concept; you can think of a method as being like a function, a subroutine, a procedure or a subprogram.

Examples of Methods

The code on the slide contains three methods:

- The **Main** method
- The **WriteLine** method
- The **ExampleMethod** method

The **Main** method is the entry point of the application. The **WriteLine** method is part of the Microsoft® .NET Framework. It can be called from within your program. The **WriteLine** method is a static method of the class **System.Console**. The **ExampleMethod** method belongs to **ExampleClass**. This method calls the **WriteLine** method.

In C#, all methods belong to a class. This is unlike programming languages such as C, C++, and Microsoft Visual Basic®, which allow global subroutines and functions.

Creating Methods

When creating a method, you must specify the following:

- Name

  You cannot give a method the same name as a variable, a constant, or any other non-method item declared in the class. The method name can be any allowable C# identifier, and it is case sensitive.

- Parameter list

  The method name is followed by a parameter list for the method. This is enclosed between parentheses. The parentheses must be supplied even if there are no parameters, as is shown in the examples on the slide.

- Body of the method

  Following the parentheses is the body of the method. You must enclose the method body within braces ({ and }), even if there is only one statement.

Syntax for Defining Methods

To create a method, use the following syntax:

```
static void MethodName( )
{
  method body
}
```

The following example shows how to create a method named **ExampleMethod** in the **ExampleClass** class:

```
using System;
class ExampleClass
{
   static void ExampleMethod( )
   {
      Console.WriteLine("Example method");
   }

   static void Main( )
   {
      Console.WriteLine("Main method");
   }
}
```

Note Method names in C# are case-sensitive. Therefore, you can declare and use methods with names that differ only in case. For example, you can declare methods called **print** and **PRINT** in the same class. However, the Common Language Runtime requires that method names within a class differ in ways other than case alone, to ensure compatibility with languages in which method names are case-insensitive. This is important if you want your application to interact with applications written in languages other than C#.

Calling Methods

- **After You Define a Method, You Can:**
  - Call a method from within the same class
    Use method's name followed by a parameter list in parentheses
  - Call a method that is in a different class
    You must indicate to the compiler which class contains the method to call
    The called method must be declared with the **public** keyword
  - Use nested calls
    Methods can call methods, which can call other methods, and so on

After you define a method, you can call it from within the same class and from other classes.

Calling Methods

To call a method, use the name of the method followed by a parameter list in parentheses. The parentheses are required even if the method that you call has no parameters, as shown in the following example.

```
MethodName( );
```

Note to Visual Basic Developers There is no **Call** statement Parentheses are required for all method calls.

In the following example, the program begins at the start of the **Main** method of **ExampleClass**. The first statement displays "The program is starting." The second statement in **Main** is the call to **ExampleMethod**. Control flow passes to the first statement within **ExampleMethod**, and " Hello, world" appears. At the end of the method, control passes to the statement immediately following the method call, which is the statement that displays "The program is ending."

```
using System;

class ExampleClass
{
   static void ExampleMethod( )
   {
      Console.WriteLine("Hello, world");
   }

   static void Main( )
   {
      Console.WriteLine("The program is starting");
      ExampleMethod( );
      Console.WriteLine("The program is ending");
   }
}
```

Calling Methods from Other Classes

To allow methods in one class to call methods in another class, you must:

- Specify which class contains the method you want to call.

  To specify which class contains the method, use the following syntax:

  ```
  ClassName.MethodName( );
  ```

- Declare the method that is called with the **public** keyword.

The following example shows how to call the method **TestMethod**, which is defined in class **A**, from **Main** in class **B**:

```
using System;

class A
{
   public static void TestMethod( )
   {
      Console.WriteLine("This is TestMethod in class A");
   }
}

class B
{
   static void Main( )
   {
      A.TestMethod( );
   }
}
```

If, in the example above, the class name were removed, the compiler would search for a method called **TestMethod** in class **B**. Since there is no method of that name in that class, the compiler will display the following error: "The name 'TestMethod' does not exist in the class or namespace 'B.'"

If you do not declare a method as public, it becomes private to the class by default. For example, if you omit the **public** keyword from the definition of **TestMethod**, the compiler will display the following error: "A.TestMethod() is inaccessible due to its protection level."

You can also use the **private** keyword to specify that the method can only be called from inside the class. The following two lines of code have exactly the same effect because methods are private by default:

```
private static void MyMethod( );
static void MyMethod( );
```

The **public** and **private** keywords shown above specify the *accessibility* of the method. These keywords control whether a method can be called from outside of the class in which it is defined.

Nesting Method Calls

You can also call methods from within methods. The following example shows how to nest method calls:

```
using System;
class NestExample
{
   static void Method1( )
   {
      Console.WriteLine("Method1")
   }
   static void Method2( )
   {
      Method1( );
      Console.WriteLine("Method2")
      Method1( );
   }
   static void Main( )
   {
      Method2( );
      Method1( );
   }
}
```

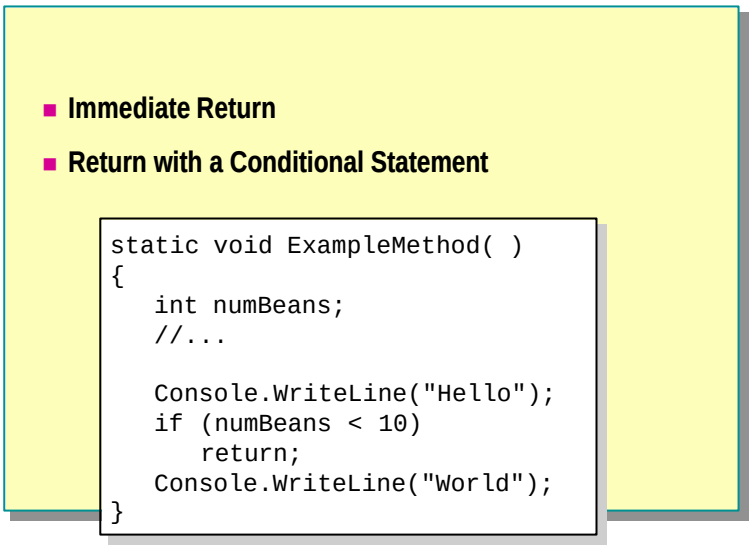
The output from this program is as follows:

```
Method1
Method2
Method1
Method1
```

You can call an unlimited number of methods by nesting. There is no predefined limit to the nesting level. However, the run-time environment might impose limits, usually because of the amount of RAM available to perform the process. Each method call needs memory to store return addresses and other information.

As a general rule, if you are running out of memory for nested method calls, you probably have a class design problem.

Using the return Statement



You can use the **return** statement to make a method return immediately to the caller. Without a **return** statement, execution usually returns to the caller when the last statement in the method is reached.

Immediate Return

By default, a method returns to its caller when the end of the last statement in the code block is reached. If you want a method to return immediately to the caller, use the **return** statement.

In the following example, the method will display " Hello," and then immediately return to its caller:

```
static void ExampleMethod( )
{
   Console.WriteLine("Hello");
   return;
   Console.WriteLine("World");
}
```

Using the **return** statement like this is not very useful because the final call to **Console.WriteLine** is never executed. If you have enabled the C# compiler warnings at level 2 or higher, the compiler will display the following message: " Unreachable code detected."

Return with a Conditional Statement

It is more common, and much more useful, to use the **return** statement as part of a conditional statement such as **if** or **switch**. This allows a method to return to the caller if a given condition is met.

In the following example, the method will return if the variable *numBeans* is less than 10; otherwise, execution will continue within this method.

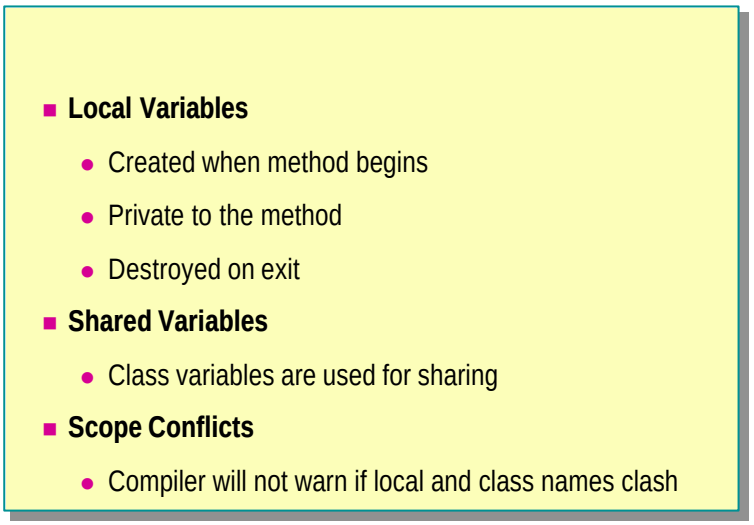
```
static void ExampleMethod( )
{
   int numBeans;
   //...
   Console.WriteLine("Hello");
   if (numBeans < 10)
      return;
   Console.WriteLine("World");
}
```

Tip It is generally regarded as good programming style for a method to have one entry point and one exit point. The design of C# ensures that all methods begin execution at the first statement. A method with no **return** statements has one exit point, at the end of the code block. A method with multiple **return** statements has multiple exit points, which can make the method difficult to understand and maintain in some cases.

Return with a Value

If a method is defined with a data type rather than **void**, the return mechanism is used to assign a value to the function. This will be discussed later in this module.

Using Local Variables



Each method has its own set of local variables. You can use these variables only inside the method in which they are declared. Local variables are not accessible from elsewhere in the application.

Local Variables

You can include local variables in the body of a method, as shown in the following example:

```
static void MethodWithLocals( )
{
   int x = 1; // Variable with initial value
   ulong y;
   string z;
}
```

You can assign local variables an initial value. (For an example, see variable *x* in the preceding code.) If you do not assign a value or provide an initial expression to determine a value, the variable will not be initialized.

The variables that are declared in one method are completely separate from variables that are declared in other methods, even if they have the same names.

Memory for local variables is allocated each time the method is called and released when the method terminates. Therefore, any values stored in these variables will not be retained from one method call to the next.

Shared Variables

Consider the following code, which attempts to count the number of times a method has been called:

```
class CallCounter_Bad
{
   static void Init( )
   {
      int nCount = 0;
   }
   static void CountCalls( )
   {
      int nCount;
      ++nCount;
      Console.WriteLine("Method called {0} time(s)", nCount);
   }
   static void Main( )
   {
      Init( );
      CountCalls( );
      CountCalls( );
   }
}
```

This program cannot be compiled because of two important problems. The variable *nCount* in **Init** is not the same as the variable *nCount* in **CountCalls**. No matter how many times you call the method **CountCalls**, the value *nCount* is lost each time **CountCalls** finishes.

The correct way to write this code is to use a class variable, as shown in the following example:

```
class CallCounter_Good
{
   static int nCount;
   static void Init( )
   {
      nCount = 0;
   }
   static void CountCalls( )
   {
      ++nCount;
      Console.Write("Method called " + nCount + " time(s).");
   }
   static void Main( )
   {
      Init( );
      CountCalls( );
      CountCalls( );
   }
}
```

In this example, *nCount* is declared at the class level rather than at the method level. Therefore, *nCount* is shared between all of the methods in the class.

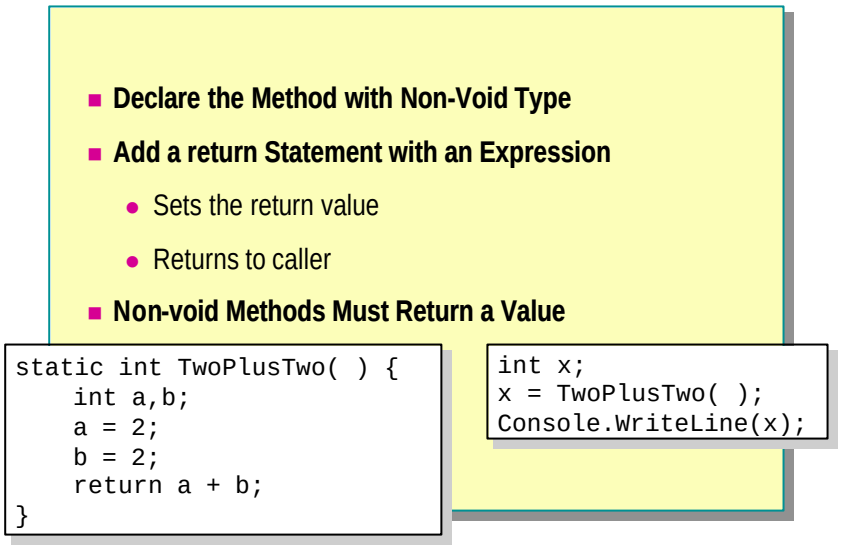
Scope Conflicts

In C#, you can declare a local variable that has the same name as a class variable, but this can produce unexpected results. In the following example, *NumItems* is declared as a variable of class **ScopeDemo**, and also declared as a local variable in **Method1**. The two variables are completely different. In **Method1**, *numItems* refers to the local variable. In **Method2**, *numItems* refers to the class variable.

```
class ScopeDemo
{
   static int numItems = 0;
   static void Method1( )
   {
       int numItems = 42;
   }
   static void Method2( )
   {
       numItems = 61;
   }
}
```

Tip Because the C# compiler will not warn you when local variables and class variables have the same names, you can use a naming convention to distinguish local variables from class variables.

Returning Values



You have learned how to use the **return** statement to immediately terminate a method. You can also use the **return** statement to return a value from a method. To return a value, you must:

1. Declare the method with the value type that you want to return.
2. Add a **return** statement inside the method.
3. Include the value that you want to return to the caller.

Declaring Methods with Non-Void Type

To declare a method so that it will return a value to the caller, replace the **void** keyword with the type of the value that you want to return.

Adding return Statements

The **return** keyword followed by an expression terminates the method immediately and returns the expression as the return value of the method.

The following example shows how to declare a method named **TwoPlusTwo** that will return a value of 4 to **Main** when **TwoPlusTwo** is called:

```
class ExampleReturningValue
{
    static int TwoPlusTwo( )
    {
        int a,b;
        a = 2;
        b = 2;
        return a + b;
    }

    static void Main( )
    {
        int x;
        x = TwoPlusTwo( );
        Console.WriteLine(x);
    }
}
```

Note that the returned value is an **int**. This is because **int** is the return type of the method. When the method is called, the value 4 is returned. In this example, the value is stored in the local variable *x* in **Main**.

Non-Void Methods Must Return Values

If you declare a method with a non-void type, you must add at least one **return** statement. The compiler attempts to check that each non-void method returns a value to the calling method in all circumstances. If the compiler detects that a non-void method has no **return** statement, it will display the following error message: "Not all code paths return a value." You will also see this error message if the compiler detects that it is possible to execute a non-void method without returning a value.

In the following example, you will get a valid return statement if the value in *x* is less than two. If the value in *x* is greater than or equal to two, the compiler will report an error because the **return** statement is not executed, and the method execution will terminate after the **if** statement without returning a value.

```
static int BadReturn( )
{
    // ...
    if (x < 2)
         return 5;
}
```

Tip You can only use the **return** statement to return one value from each method call. If you need to return more than one value from a method call, you can use the **ref** or **out** parameters, which are discussed later in this module. Alternatively, you can return a reference to an array or class, which can contain multiple values. The general guideline that says to avoid using multiple **return** statements in a single method applies equally to non-void methods.

Using Parameters

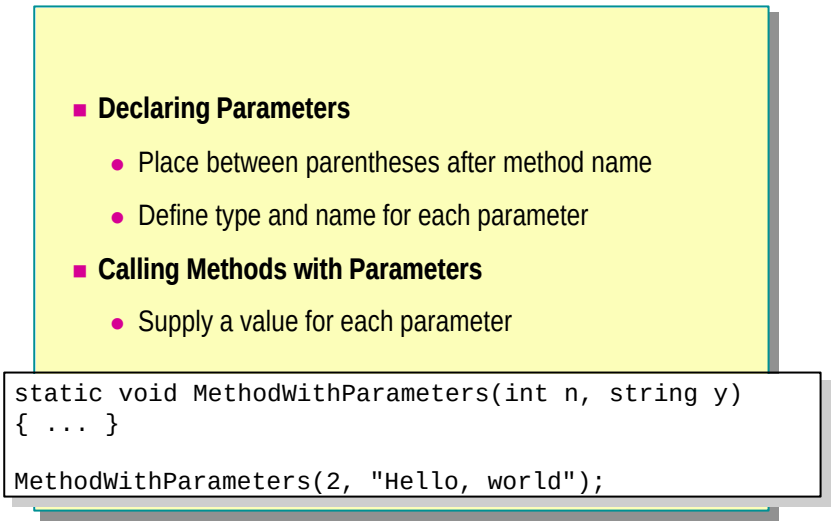
- Declaring and Calling Parameters
- Mechanisms for Passing Parameters
- Pass by Value
- Pass by Reference
- Output Parameters
- Using Variable-Length Parameter Lists
- Guidelines for Passing Parameters
- Using Recursive Methods

In this section, you will learn how to declare parameters and how to call methods with parameters. You will also learn how to pass parameters. Finally, you will learn how C# supports recursive method calls.

In this section you will learn how to:

- Declare and call parameters.
- Pass parameters by using the following mechanisms:
  - Pass by value
  - Pass by reference
  - Output parameters
- Use recursive method calls.

Declaring and Calling Parameters



Parameters allow information to be passed into and out of a method. When you define a method, you can include a list of parameters in parentheses following the method name. In the examples so far in this module, the parameter lists have been empty.

Declaring Parameters

Each parameter has a type and a name. You declare parameters by placing the parameter declarations inside the parentheses that follow the name of the method. The syntax that is used to declare parameters is similar to the syntax that is used to declare local variables, except that you separate each parameter declaration with a comma instead of with a semicolon.

The following example shows how to declare a method with parameters:

```
static void MethodWithParameters(int n, string y)
{
    // ...
}
```

This example declares the **MethodWithParameters** method with two parameters: *n* and *y*. The first parameter is of type **int**, and the second is of type **string**. Note that commas separate each parameter in the parameter list.

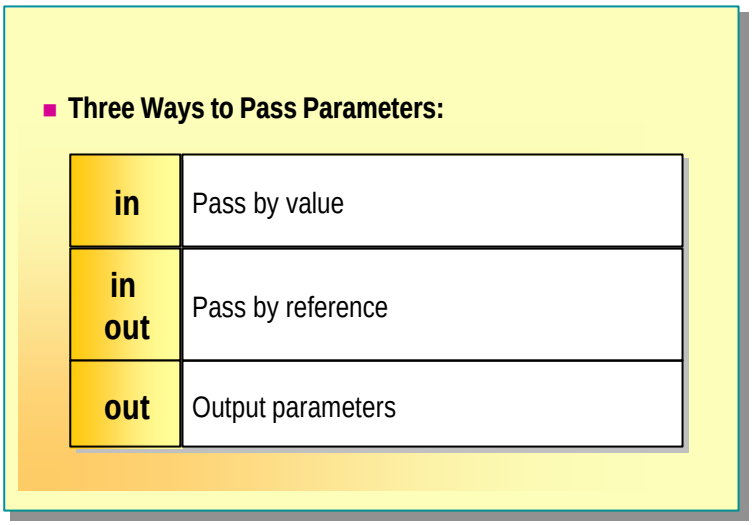
Calling Methods with Parameters

The calling code must supply the parameter values when the method is called.

The following code shows two examples of how to call a method with parameters. In each case, the values of the parameters are found and placed into the parameters *n* and *y* at the start of the execution of **MethodWithParameters**.

```
MethodWithParameters(2, "Hello, world");

int p = 7;
string s = "Test message";

MethodWithParameters(p, s);
```

Mechanisms for Passing Parameters



Parameters can be passed in three different ways:

- By value

  Value parameters are sometimes called *in parameters* because data can be transferred into the method but cannot be transferred out.

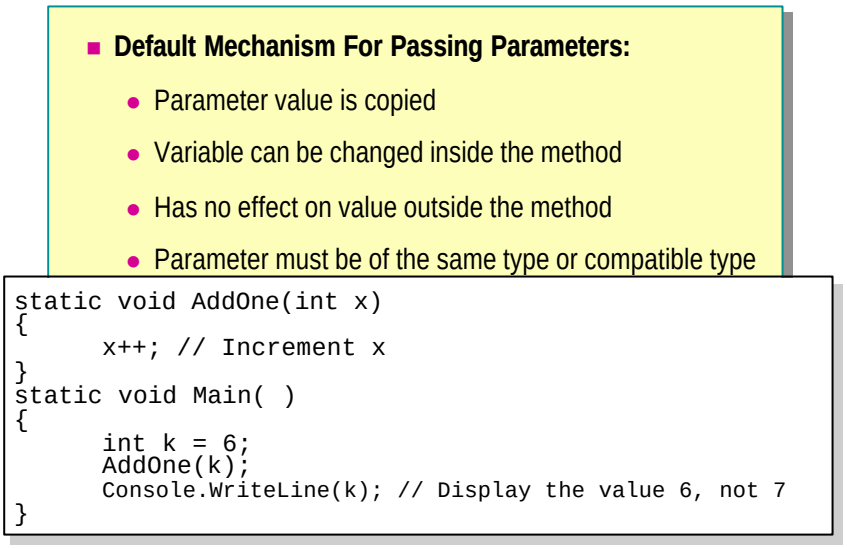
- By reference

  Reference parameters are sometimes called *in/out parameters* because data can be transferred into the method and out again.

- By output

  Output parameters are sometimes called *out parameters* because data can be transferred out of the method but cannot be transferred in.

Pass by Value



In most applications, most parameters are used for passing information into a method but not out. Therefore, pass by value is the default mechanism for passing parameters in C#.

Defining Value Parameters

The simplest definition of a parameter is a type name followed by a variable name. This is known as a *value parameter*. When the method is called, a new storage location is created for each value parameter, and the values of the corresponding expressions are copied into them.

The expression supplied for each value parameter must be the same type as the declaration of the value parameter, or a type that can be implicitly converted to that type. Within the method, you can write code that changes the value of the parameter. It will have no effect on any variables outside the method call.

In the following example, the variable *x* inside **AddOne** is completely separate from the variable *k* in **Main**. The variable *x* can be changed in **AddOne**, but this has no effect on *k*.

```
static void AddOne(int x)
{
    x++;
}
static void Main( )
{
    int k = 6;
    AddOne(k);
    Console.WriteLine(k); // Display the value 6, not 7
}
```

Pass by Reference

- **What Are Reference Parameters?**
  - A reference to memory location
- **Using Reference Parameters**
  - Use the **ref** keyword in method declaration and call
  - Match types and variable values
  - Changes made in the method affect the caller
  - Assign parameter value before calling the method

What Are Reference Parameters?

A reference parameter is a reference to a memory location. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same location in memory as the variable that is supplied in the method call.

Declaring Reference Parameters

You can declare a reference parameter by using the **ref** keyword before the type name, as shown in the following example:

```
static void ShowReference(ref int nVar, ref long nCount)
{
    // ...
}
```

Using Multiple Parameter Types

The **ref** keyword only applies to the parameter following it, not to the whole parameter list. Consider the following method, in which *refVar* is passed by reference but *longVar* is passed by value:

```
static void OneRefOneVal(ref int refVar, long longVar)
{
  // ...
}
```

Matching Parameter Types and Values

When calling the method, you supply reference parameters by using the **ref** keyword followed by a variable. The value supplied in the call to the method must exactly match the type in the method definition, and it must be a variable, not a constant or calculated expression.

```
int x;
long q;
ShowReference(ref x, ref q);
```

If you omit the **ref** keyword, or if you supply a constant or calculated expression, the compiler will reject the call, and you will receive an error message similar to the following: "Cannot convert from 'int' to 'ref int.'"

Changing Reference Parameter Values

If you change the value of a reference parameter, the variable supplied by the caller is also changed, because they are both references to the same location in memory. The following example shows how changing the reference parameter also changes the variable:

```
static void AddOne(ref int x)
{
    x++;
}
static void Main( )
{
    int k = 6;
    AddOne(ref k);
    Console.WriteLine(k); // Display the value 7
}
```

This works because when **AddOne** is called, its parameter *x* is set up to refer to the same memory location as the variable *k* in **Main**. Therefore, incrementing *x* will increment *k*.

Assigning Parameters Before Calling the Method

A **ref** parameter must be definitively assigned at the point of call; that is, the compiler must ensure that a value is assigned before the call is made. The following example shows how you can initialize reference parameters before calling the method:

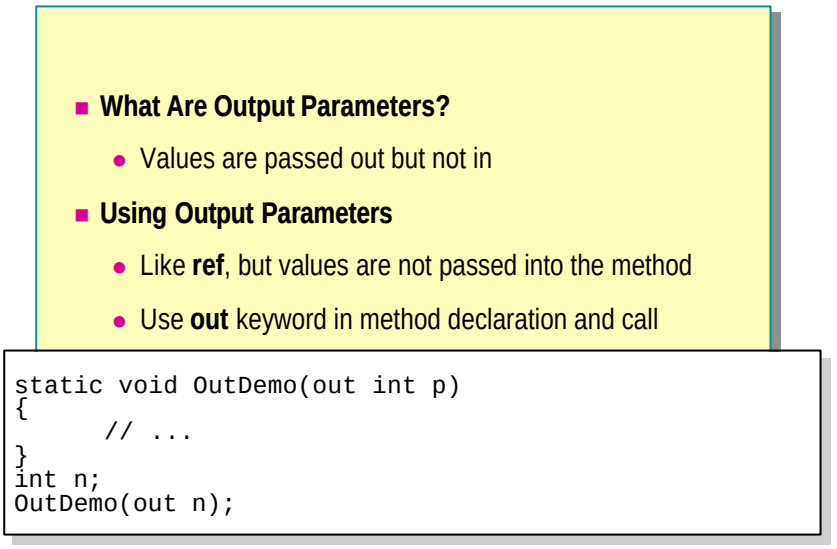
```
static void AddOne(ref int x)
{
    x++;
}

static void Main( )
{
    int k = 6;
    AddOne(ref k);
    Console.WriteLine(k); // 7
}
```

The following example shows what happens if a reference parameter *k* is not initialized before its method **AddOne** is called:

```
int k;
AddOne(ref k);
Console.WriteLine(k);
```

The C# compiler will reject this code and display the following error message: " Use of unassigned local variable '*k*.'"

Output Parameters



What Are Output Parameters?

Output parameters are like reference parameters, except that they transfer data out of the method rather than into it. They are similar to reference parameters. Like a reference parameter, an output parameter is a reference to a storage location supplied by the caller. However, the variable that is supplied for the **out** parameter does not need to be assigned a value before the call is made, and the method will assume that the parameter has not been initialized on entry.

Output parameters are useful when you want to be able to return values from a method by means of a parameter without assigning an initial value to the parameter.

Using Output Parameters

To declare an output parameter, use the keyword **out** before the type and name, as shown in the following example:

```
static void OutDemo(out int p)
{
    // ...
}
```

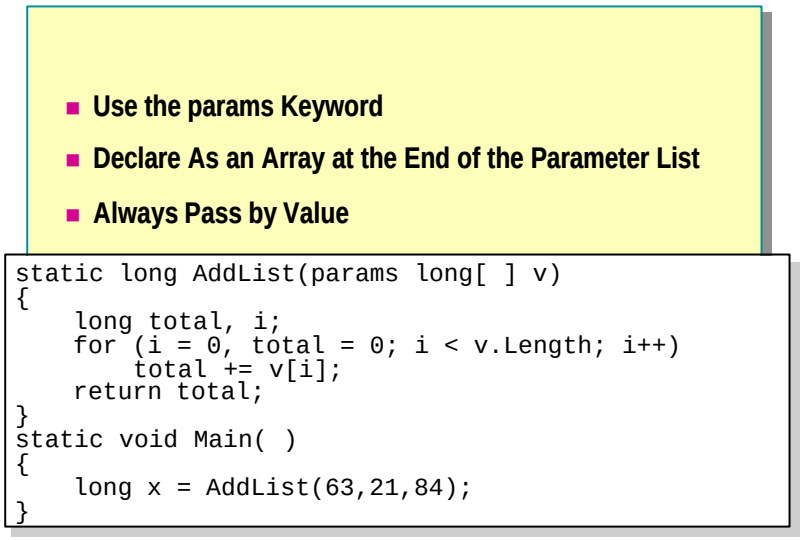
As with the **ref** keyword, the **out** keyword only affects one parameter, and each **out** parameter must be marked separately.

When calling a method with an **out** parameter, place the **out** keyword before the variable to be passed, as in the following example.

```
int n;
OutDemo(out n);
```

In the body of the method being called, no initial assumptions are made about the contents of the output parameter. It is treated just like an unassigned local variable.

Using Variable-Length Parameter Lists



C# provides a mechanism for passing variable-length parameter lists.

Declaring Variable-Length Parameters

It is sometimes useful to have a method that can accept a varying number of parameters. In C#, you can use the **params** keyword to specify a variable-length parameter list. When you declare a variable-length parameter, you must:

- Declare only one **params** parameter per method.
- Place the parameter at the end of the parameter list.
- Declare the parameter as a single-dimension array type.

The following example shows how to declare a variable-length parameter list:

```
static long AddList(params long[ ] v)
{
    long total;
    long i;
    for (i = 0, total = 0; i < v. Length; i++)
        total += v[i];
    return total;
}
```

Because a **params** parameter is always an array, all values must be the same type.

Passing Values

When you call a method with a variable-length parameter, you can pass values to the **params** parameter in one of two ways:

- As a list of elements (the list can be empty)
- As an array

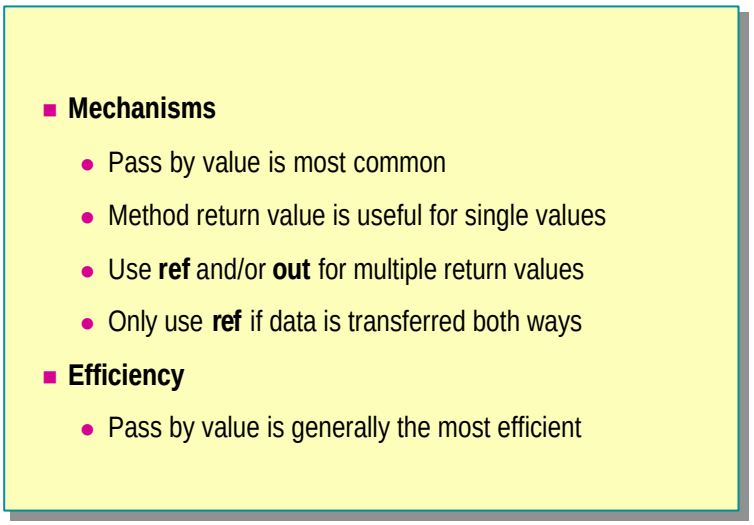
The following code shows both techniques. The two techniques are treated in exactly the same way by the compiler.

```
static void Main( )
{
    long x;
    x = AddList(63,21,84); // List
    x = AddList(new long[ ]{ 63, 21, 84 }); // Array
}
```

Regardless of which method you use to call the method, the **params** parameter is treated like an array. You can use the **Length** property of the array to determine how many parameters were passed to each call.

In a **params** parameter, a copy of the data is made, and although you can modify the values inside the method, the values outside the method are unchanged.

Guidelines for Passing Parameters



With so many options available for parameter passing, the most appropriate choice might not be obvious. Two factors for you to consider when you choose a way to pass parameters are the mechanism and its efficiency.

Mechanisms

Value parameters offer a limited form of protection against unintended modification of parameter values, because any changes that are made inside the method have no effect outside it. This suggests that you should use value parameters unless you need to pass information out of a method.

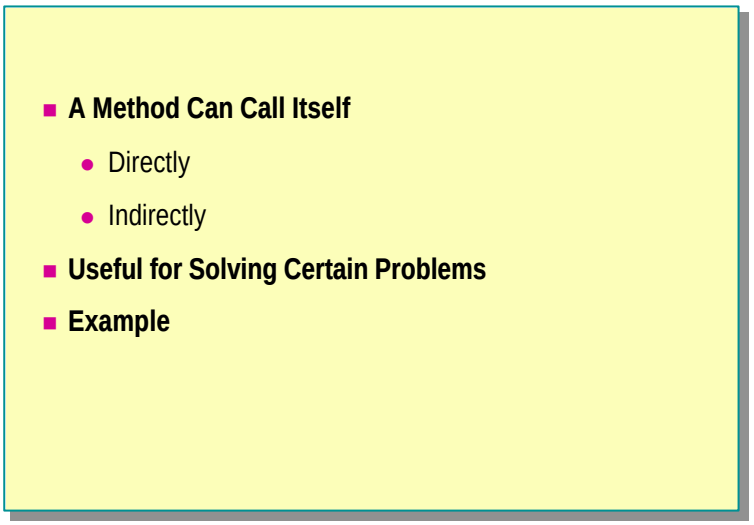
If you need to pass data out of a method, you can use the **return** statement, reference parameters, or output parameters. The **return** statement is easy to use, but it can only return one result. If you need multiple values returned, you must use the reference and output parameter types. Use **ref** if you need to transfer data in both directions, and use **out** if you only need to transfer data out of the method.

Efficiency

Generally, simple types such as int and long are most efficiently passed by value.

These efficiency concerns are not built into the language, and you should not rely on them. Although efficiency is sometimes a consideration in large, resource-intensive applications, it is usually better to consider program correctness, stability, and robustness before efficiency. Make good programming practices a higher priority than efficiency.

Using Recursive Methods



A method can call itself. This technique is known as *recursion*. You can address some types of problems with recursive solutions. Recursive methods are often useful when manipulating more complex data structures such as lists and trees.

Methods in C# can be mutually recursive. For example, a situation in which method A can call method B, and method B can call method A, is allowable.

Example of a Recursive Method

The Fibonacci sequence occurs in several situations in mathematics and biology (for example, the reproductive rate and population of rabbits). The *n*th member of this sequence has the value 1 if *n* is 1 or 2; otherwise, it is equal to the sum of the preceding two numbers in the sequence. Notice that when *n* is greater than two the value of the *n*th member of the sequence is derived from the values of two previous values of the sequence. When the definition of a method refers to the method itself, recursion might be involved.

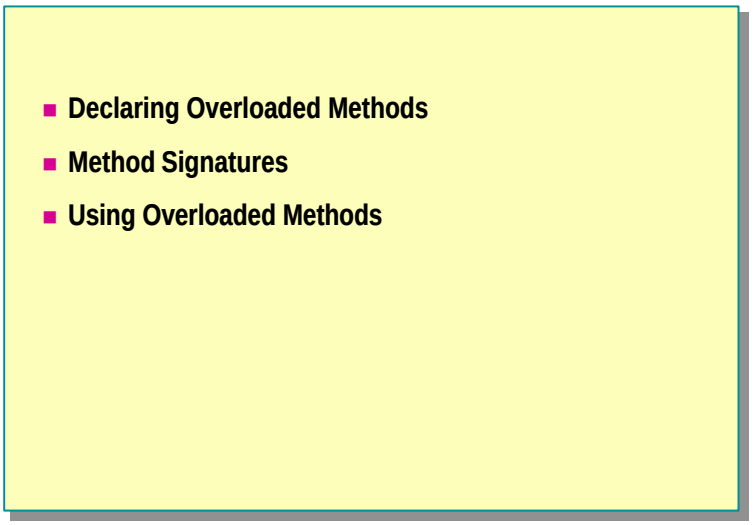
You can implement the **Fibonacci** method as follows:

```
static ulong Fibonacci(ulong n)
{
    if (n <= 2)
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Notice that two calls are made to the method from within the method itself.

A recursive method must have a terminating condition that ensures that it will return without making further calls. In the case of the **Fibonacci** method, the test for `n <= 2` is the terminating condition.

Using Overloaded Methods

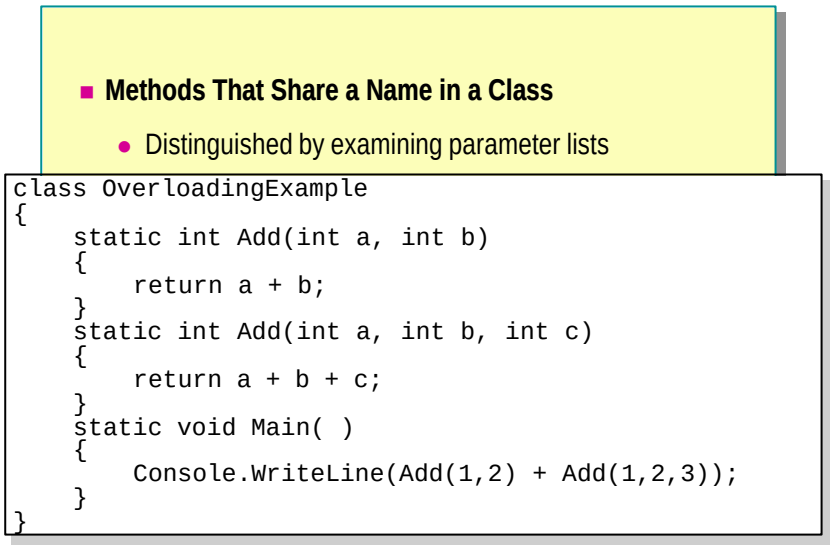


Methods might not have the same name as other non-method items in a class. However, it is possible for two or more methods in a class to share the same name. Name sharing among methods is called overloading.

In this section, you will learn:

- How to declare overloaded methods.
- How C# uses signatures to distinguish methods that have the same name.
- When to use overloaded methods.

Declaring Overloaded Methods



Overloaded methods are methods in a single class that have the same name. The C# compiler distinguishes overloaded methods by comparing the parameter lists.

Examples of Overloaded Methods

The following code shows how you can use different methods with the same name in one class:

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(3,4,5));
    }
}
```

The C# compiler finds two methods called **Add** in the class, and two method calls to methods called **Add** within **Main**. Although the method names are the same, the compiler can distinguish between the two **Add** methods by comparing the parameter lists.

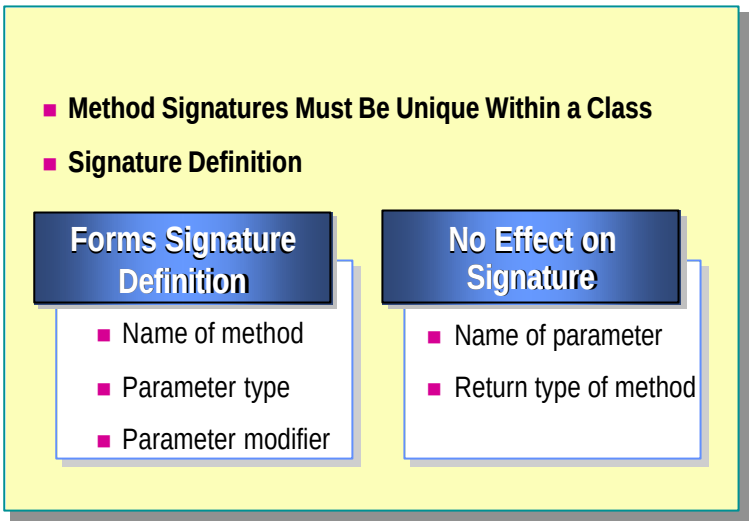
The first **Add** method takes two parameters, both of type **int**. The second **Add** method takes three parameters, also of type **int**. Because the parameter lists are different, the compiler allows both methods to be defined within the same class.

The first statement within **Main** includes a call to **Add** with two **int** parameters, so the compiler translates this as a call to the first **Add** method. The second call to **Add** takes three **int** parameters, so the compiler translates this as a call to the second **Add** method.

You cannot share names among methods and variables, constants, or enumerated types in the same class. The following code will not compile because the name *k* has been used for both a method and a class variable:

```
class BadMethodNames
{
  static int k;
  static void k( ) {
    // ...
  }
}
```

Method Signatures



The C# compiler uses signatures to distinguish between methods in a class. In each class, the signature of each method must differ from the signatures of all other methods that are declared in that class.

Signature Definition

The signature of a method consists of the name of the method, the number of parameters that the method takes, and the type and modifier (such as **out** or **ref**) of each parameter.

The following three methods have different signatures, so they can be declared in the same class.

```
static int LastErrorCode( )
{

}
static int LastErrorCode(int n)
{

}

static int LastErrorCode(int n, int p)
{

}
```

Elements That Do Not Affect the Signature

The method signature does *not* include the return type. The following two methods have the same signatures, so they cannot be declared in the same class.

```
static int LastErrorCode(int n)
{
}
static string LastErrorCode(int n)
{
}
```

The method signature does *not* include the names of the parameters. The following two methods have the same signature, even though the parameter names are different.

```
static int LastErrorCode(int n)
{
}
static int LastErrorCode(int x)
{
}
```

Using Overloaded Methods

- **Consider Using Overloaded Methods When:**
  - You have similar methods that require different parameters
  - You want to add new functionality to existing code
- **Do Not Overuse Because:**
  - Hard to debug
  - Hard to maintain

Overloaded methods are useful when you have two similar methods that require different numbers or types of parameters.

Similar Methods That Require Different Parameters

Imagine that you have a class containing a method that sends a greeting message to the user. Sometimes the user name is known, and sometimes it is not. You could define two different methods called **Greet** and **GreetUser**, as shown in the following code:

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Hello");
    }
    static void GreetUser(string Name)
    {
        Console.WriteLine("Hello" + Name);
    }
    static void Main( )
    {
        Greet( );
        GreetUser("Alex");
    }
}
```

This will work, but now the class has two methods that perform almost exactly the same task but that have different names. You can rewrite this class with method overloading as shown in the following code:

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Hello");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Hello" + Name);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
    }
}
```

Adding New Functionality to Existing Code

Method overloading is also useful when you want to add new features to an existing application without making extensive changes to existing code. For example, the previous code could be expanded by adding another method that greets a user with a particular greeting, depending on the time of day, as shown in the following code:

```
class GreetDemo
{
    enum TimeOfDay { Morning, Afternoon, Evening }

    static void Greet( )
    {
        Console.WriteLine("Hello");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Hello" + Name);
    }
    static void Greet(string Name, TimeOfDay td)
    {
        string Message;

        switch(td)
        {
        case TimeOfDay.Morning:
            Message="Good morning";
            break;
        case TimeOfDay.Afternoon:
            Message="Good afternoon";
            break;
        case TimeOfDay.Evening:
            Message="Good evening";
            break;
        }
        Console.WriteLine(Message + " " + Name);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
        Greet("Sandra", TimeOfDay.Morning);
    }
}
```

Determining When to Use Overloading

Overuse of method overloading can make classes hard to maintain and debug. In general, only overload methods that have very closely related functions but differ in the amount or type of data that they need.

Lab 5: Creating and Using Methods

Objectives

After completing this lab, you will be able to:

- Create and call methods with and without parameters.
- Use various mechanisms for passing parameters.

Prerequisites

Before working on this lab, you should be familiar with the following:

- Creating and using variables
- C# statements

Estimated time to complete this lab: 30 minutes

Exercise 1
Using Parameters in Methods That Return Values

In this exercise, you will define and use input parameters in a method that returns a value. You will also write a test framework to read two values from the console and display the results.

You will create a class called **Utils**. In this class, you will create a method called **Greater**. This method will take two integer parameters as input and will return the value of the greater of the two.

To test the class, you will create another class called **Test** that prompts the user for two numbers, then calls **Utils.Greater** to determine which number is the greater of the two, and then prints the result.

↙ To create the Greater method

1. Open the Utils.sln project in the *install folder*\Labs\Lab05\Starter\Utility folder.

   This contains a namespace called **Utils** that contains a class also called **Utils**. You will write the **Greater** method in this class.

2. Create the **Greater** method as follows:

   a. Open the **Utils** class.

   b. Add a public static method called **Greater** to the **Utils** class.

   c. The method will take two **int** parameters, called *a* and *b*, which will be passed by value. The method will return an **int** value representing the greater of the two numbers.

The code for the **Utils** class should be as follows:

```
namespace Utils
{
    using System;

    class Utils
    {

       //
       // Return the greater of two integer values
       //

       public static int Greater(int a, int b)
       {
         if (a > b)
             return a;
         else
             return b;
       }
    }
}
```

↙ To test the Greater method

1. Open the **Test** class.
2. Within the **Main** method, write the following code.
   a. Define two integer variables called *x* and *y*.
   b. Add statements that read two integers from keyboard input and use them to populate *x* and *y*. Use the **Console.ReadLine** and **int.Parse** methods that were presented in earlier modules.
   c. Define another integer called *greater*.
   d. Test the **Greater** method by calling it, and assign the returned value to the variable *greater*.
3. Write code to display the greater of the two integers by using **Console.WriteLine**.

The code for the **Test** class should be as follows:

```
namespace Utils
{
    using System;

    /// <summary>
    ///   This the test harness
    /// </summary>

    public class Test
    {
        public static void Main( )
        {
            int x;       // Input value 1
            int y;       // Input value 2
            int greater; // Result from Greater( )

            // Get input numbers
            Console.WriteLine("Enter first number:");
            x = int.Parse(Console.ReadLine( ));
            Console.WriteLine("Enter second number:");
            y = int.Parse(Console.ReadLine( ));

            // Test the Greater( ) method
            greater = Utils.Greater(x,y);
            Console.WriteLine("The greater value is "+
➥ greater);

        }
    }
}
```

4. Save your work.
5. Compile the project and correct any errors. Run and test the program.

Exercise 2
Using Methods with Reference Parameters

In this exercise, you will write a method called **Swap** that will exchange the values of its parameters. You will use parameters that are passed by reference.

↙ To create the Swap method

1. Open the Utils.sln project in the *install folder*\Labs\Lab05\Starter\Utility folder, if it is not already open.
2. Add the **Swap** method to the **Utils** class as follows:
   a. Add a public static void method called **Swap**.
   b. **Swap** will take two **int** parameters called *a* and *b*, which will be passed by reference.
   c. Write statements inside the body of **Swap** that exchange the values of *a* and *b*. You will need to create a local **int** variable in **Swap** to temporarily hold one of the values during the exchange. Name this variable *temp*.

   The code for the **Utils** class should be as follows:

```
namespace Utils
{
    using System;

    public class Utils
    {

    ... existing code omitted for clarity ...

    //
    // Exchange two integers, passed by reference
    //

    public static void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }

    }
}
```

↙ To test the Swap method

1. Edit the **Main** method in the **Test** class by performing the following steps:
   a. Populate integer variables *x* and *y*.
   b. Call the **Swap** method, passing these values as parameters.

   Display the new values of the two integers before and after exchanging them. The code for the **Test** class should be as follows:

```
namespace Utils
{
    using System;

    public class Test
    {

    public static void Main( )
    {
        ... existing code omitted for clarity ...

        // Test the Swap method
        Console.WriteLine("Before swap: " + x + "," + y);
        Utils.Swap(ref x,ref y);
        Console.WriteLine("After swap: " + x + "," + y);

    }

    }
}
```

2. Save your work.
3. Compile the project, correcting any errors you find. Run and test the program.

Tip If the parameters were not exchanged as you expected, check to ensure that you passed them as **ref** parameters.

Exercise 3
Using Methods with Output Parameters

In this exercise, you will define and use a static method with an output parameter.

You will write a new method called **Factorial** that takes an **int** value and calculates its factorial. The factorial of a number is the product of all the numbers between 1 and that number. The factorial of zero is defined to be 1. The following are examples of factorials:

- Factorial(0) = 1
- Factorial(1) = 1
- Factorial(2) = 1 * 2 = 2
- Factorial(3) = 1 * 2 * 3 = 6
- Factorial(4) = 1 * 2 * 3 * 4 = 24

↙ To create the Factorial method

1. Open the Utils.sln project in the *install folder*\Labs\Lab05\Starter\Utility folder, if it is not already open.
2. Add the **Factorial** method to the **Utils** class, as follows:
   a. Add a new public static method called **Factorial**.
   b. This method will take two parameters called *n* and *answer*. The first, passed by value, is an **int** value for which the factorial is to be calculated. The second parameter is an **out int** parameter that will be used to return the result.
   c. The **Factorial** method should return a **bool** value that indicates whether the method succeeded. (It could overflow and raise an exception.)
3. Add functionality to the **Factorial** method.

   The easiest way to calculate a factorial is by using a loop. Perform the following steps to add functionality to the method:

   a. Create an **int** variable called *k* in the **Factorial** method. This will be used as a loop counter.
   b. Create another **int** variable called *f*, which will be used as a working value inside the loop. Initialize the working variable *f* with the value 1.
   c. Use a **for** loop to perform the iteration. Start with a value of 2 for *k*, and finish when *k* reaches the value of parameter *n*. Increment *k* each time the loop is performed.
   d. In the body of the loop, multiply *f* successively by each value of *k*, storing the result in *f*.
   e. Factorial results can be very large even for small input values, so ensure that all the integer calculations are in a checked block, and that you have caught exceptions such as arithmetic overflow.
   f. Assign the result value in *f* to the out parameter *answer*.
   g. Return **true** from the method if the calculation is successful, and **false** if the calculation is not successful (that is, if an exception occurs).

The code for the **Utils** class should be as follows:

```
namespace Utils
{
    using System;

    public class Utils
    {

    ... existing code omitted for clarity ...

    //
    // Calculate factorial
    // and return the result as an out parameter
    //

    public static bool Factorial(int n, out int answer)
    {
        int k;        // Loop counter
        int f;        // Working value
        bool ok=true; // True if okay, false if not

        // Check the input value

        if (n<0)
            ok = false;

        // Calculate the factorial value as the
        // product of all of the numbers from 2 to n

        try
        {
            checked
            {
                f = 1;
                for (k=2; k<=n; ++k)
                {
                    f = f * k;
                }
            }
        }
        catch(Exception)
        {
            // If something goes wrong in the calculation,
            // catch it here. All exceptions
            // are handled the same way: set the result
            // to zero and return false.
```

(Code continued on following page.)

```
                f = 0;
                ok = false;
            }


            // Assign result value
            answer = f;
            // Return to caller
            return ok;
        }


    }
}
```

↙ To test the Factorial method

1. Edit the **Test** class as follows:
   a. Declare a **bool** variable called *ok* to hold the **true** or **false** result.
   b. Declare an **int** variable called *f* to hold the factorial result.
   c. Request an integer from the user. Assign the input value to the **int** variable *x*.
   d. Call the **Factorial** method, passing *x* as the first parameter and *f* as the second parameter. Return the result in *ok*.
   e. If *ok* is **true**, display the values of *x* and *f*; otherwise, display a message indicating that an error has occurred.

   The code for the **Test** class should be as follows:

```
namespace Utils
{
    public class Test
    {

    static void Main( )
    {
        int f;       // Factorial result
        bool ok;     // Factorial success or failure

        ... existing code omitted for clarity ...

        // Get input for factorial

        Console.WriteLine("Number for factorial:");
        x = int.Parse(Console.ReadLine( ));

        // Test the factorial function
        ok = Utils.Factorial(x, out f);
        // Output factorial results
        if (ok)
            Console.WriteLine("Factorial(" + x + ") = " +
f);
        else
            Console.WriteLine("Cannot compute this
➥factorial");
    }
    }
}
```

2. Save your work.
3. Compile the program, correct any errors, and then run and test the program.

If Time Permits
Implementing a Method by Using Recursion

In this exercise, you will re-implement the **Factorial** method that you created in Exercise 3 by using recursion rather than a loop.

The factorial of a number can be defined recursively as follows: the factorial of zero is 1, and you can find the factorial of any larger integer by multiplying that integer with the factorial of the previous number. In summary:

If n=0, then Factorial(n) = 1; otherwise it is n * Factorial(n-1)

↙ To modify the existing Factorial method

1. Edit the **Utils** class and modify the existing **Factorial** method so that it uses recursion rather than iteration.

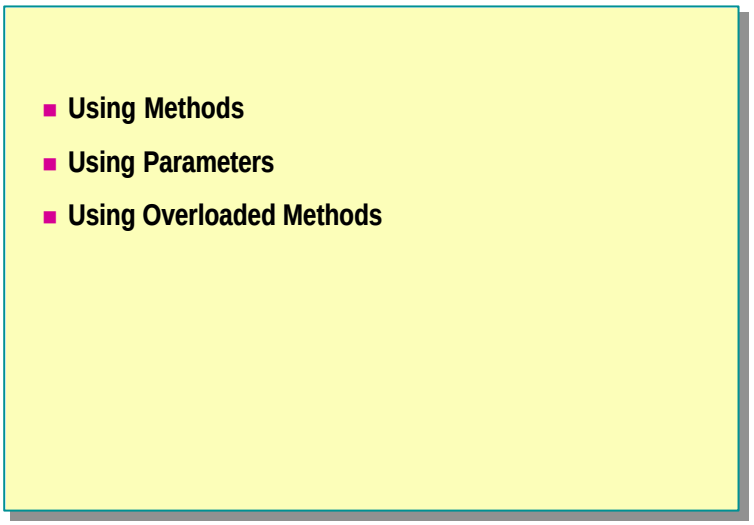
   The parameters and return types will be the same, but the internal functionality of the method will be different. If you want to keep your existing solution to Exercise 3, you will need to use another name for this method.

2. Use the pseudocode shown above to implement the body of the **Factorial** method. (You will need to convert it into C# syntax.)
3. Add code to the **Test** class to test your new method.
4. Save your work.
5. Compile the program, correct any errors, and then run and test the program.

The recursive version of the **Factorial** method (**RecursiveFactorial**) is shown below:

```
//
// Another way to solve the factorial problem,
// this time as a recursive function
//

public static bool RecursiveFactorial(int n, out int f)
{
    bool ok=true;

    // Trap negative inputs
    if (n<0)
    {
            f=0;
            ok = false;
    }

    if (n<=1)
        f=1;
    else
    {
        try
        {
            int pf;
            checked
            {
                ok = RecursiveFactorial(n-1,out pf);
                f = n * pf;
            }
        }
        catch(Exception)
        {
            // Something went wrong. Set error
            // flag and return zero.
            f=0;
            ok=false;
        }

    }

    return ok;
}
```

Review



1. Explain what methods are and why they are important.

2. List the three ways in which data can be passed in parameters, and the associated C# keywords.

3. When are local variables created and destroyed?

4. What keyword should be added to a method definition if the method needs to be called from another class?

5. What parts of a method are used to form the signature?

6. Define the signature of a static method called **Rotate** that does not return a value but that must "right rotate" its three integer parameters.