**Declaring Methods**

The ability to define and call methods is a fundamental component of object-oriented programming, because methods enable you to encapsulate operations that protect data that is stored inside a type.

Typically, any application that you develop by using the Microsoft .NET Framework and Visual C# will have many methods, each with a specific purpose. Some methods are fundamental to the operation of an application. For example, all Visual C# desktop applications must have a method called **Main** that defines the entry point for the application. When the user runs a Visual C# application, the common language runtime (CLR) executes the **Main** method for that application.

Methods can be designed for internal use by a type, and as such are hidden from other types. Public methods may be designed to enable other types to request that an object performs an action, and are exposed outside of the type.

The .NET Framework itself is built from classes that expose methods that you can call from your applications to interact with the user and the computer.

A method is declared using a method signature and method body.   The signature portion is responsible for providing the access modifier, method return type, the method name, and the list of parameters.  The body contains the implementation for what the method is intended to do.   Each method signature component is explained here:

* Access modifier - this is used to control the accessibility of the method (from where it can be called)
  + private - most restrictive and allows access to the method only from within the containing class or struct
  + public - least restrictive, allowing access from any code in the application
  + protected - allows for access from within the containing class or from within derived classes
  + internal - accessible from files within the same assembly
  + static - indicates the method is a static member of the class rather than a member of an instance of a specific object
* Return type - used to indicate what type the method will return.  Use void if the method will not return a value or any supported data type
* Method name - all methods need a name so you know what to call in code.  Identifier rules apply to methods names as well
* Parameter list - a comma separated list of parameters to accept arguments passed into the method

Sample method:

public Boolean StartService(string serviceName)  
{  
   // code to start the service  
}

In the preceding example, public is the access modifier, Boolean is the return type, StartService is the name, and string serviceName is the parameter list.  Note that the parameter list specifies a data type and a name for the parameter.

**Calling a Method**

You call a method to run the code in that method from part of your application. You do not need to understand how the code in a method works. You may not even have access to the code, if it is in a class in an assembly for which you do not have the source, such as the .NET Framework class library.

To call a method, you specify the method name and provide any arguments that correspond to the method parameters in brackets.

The following code example shows how to invoke the **StartService** method, passing **int** and **Boolean** variables to satisfy the parameter requirements of the method’s signature.

var upTime = 2000;  
var shutdownAutomatically = true;  
StartService(upTime, shutdownAutomatically);    
        
// StartService method.  
void StartService(int upTime, bool shutdownAutomatically)  
{  
   // Perform some processing here.  
}

**Returning Data**

If the method returns a value, you specify how to handle this value, typically by assigning it to a variable of the same type, in your calling code.

The following code example shows how to capture the return value of the **GetServiceName** method in a variable named **serviceName**.

string serviceName = GetServiceName();  
string GetServiceName()  
{  
   return "FourthCoffee.SalesService";  
}

The above example shows returning a single value from the method.   There may be times when you would prefer to return multiple values from a method.  There are three approaches that you can take to accomplish this:

* Return an array or collection
* Use the ref keyword
* Use the out keyword

In this first code example, a call to the method ReturnMultiOut is made.  The parameters for this method use the out keyword to indicate that values will be returned for these parameters.  Note that we do not have to call this method with an assignment statement as in the previous method call to GetServiceName()  
  
ReturnMultiOut(out first, out sValue);  
Console.WriteLine("{0}, {1}", first.ToString(), sValue);  
  
static void ReturnMultiOut(out int i, out string s)  
{  
    i = 25;  
    s = "using out";  
}

In this code example, the keyword ref is used to return multiple values from the method.  Typically the refkeyword requires that the variables being used are initialized first.  
  
// Using ref requires that the variables be initialized first  
sValue = "";  
ReturnMultiRef(ref first, ref sValue);  
Console.WriteLine("{0}, {1}", first.ToString(), sValue);  
  
 static void ReturnMultiRef(ref int i, ref string s)  
 {  
        i = 50;  
        s = "using ref";  
 }

**Overloading Methods**

When you define a method, you might realize that it requires different sets of information in different circumstances. You can define overloaded methods to create multiple methods with the same functionality that accept different parameters depending on the context in which they are called.

Overloaded methods have the same name as each other to emphasize their common intent. However, each overloaded method must have a unique signature, to differentiate it from the other overloaded versions of the method in the class.

The signature of a method includes its name and its parameter list. The return type is not part of the signature. Therefore, you cannot define overloaded methods that differ only in their return type.  You can also not define overloaded methods that differ in position of the parameters.

The following code example shows three versions of the **StopService** method, all with a unique signature.

void StopService()  
{  
   // This method accepts no arguments  
}  
void StopService(string serviceName)  
{  
   // This method overload accepts a single string argument  
}  
void StopService(int serviceId)  
{  
   // This method overload accepts a single integer argument  
}

When you invoke the **StopService** method, you have choice of which overloaded version you use. You simply provide the relevant arguments to satisfy a particular overload, and then the compiler works out which version to invoke based on the arguments that you passed.

**Optional Parameters**

A key feature of Visual C# is the ability to interoperate with applications and components that are written by using other technologies. One of the principal technologies that Windows uses is the Component Object Model (COM). COM does not support overloaded methods, but instead uses methods that can take optional parameters. To make it easier to incorporate COM libraries and components into a Visual C# solution, Visual C# also supports optional parameters.

Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations.   When defining methods that will use optional parameters, it's important to note that you must specify all non-optional parameters first, and then list the optional parameters.

The following code example shows how to define a method that accepts one mandatory parameter (forceStop) and two optional parameters (serviceName, serviceID).  Note that the mechanism used to denote an optional parameter is the inclusion if a default value.

void StopService(bool forceStop, string serviceName = null, int serviceId =1)  
{  
   // code here that will stop the service  
}  
  
You can call a method that takes optional parameters in the same way that you call any other method. You specify the method name and provide any necessary arguments. The difference with methods that take optional parameters is that you can omit the corresponding arguments, and the method will use the default value when the method runs.

**Named Parameters**

Traditionally, when calling a method, the order and position of arguments in the method call corresponds to the order of parameters in the method signature. If the arguments are misaligned and the types mismatched, you receive a compile error.

In Visual C#, you can specify parameters by name, and therefore supply arguments in a sequence that differs from that defined in the method signature. To use named arguments, you supply the parameter name and corresponding value separated by a colon.

The following code example shows how to invoke the **StopService**method by using named arguments to pass the **serviceID** parameter.

StopService(true, serviceID: 1);

When using named arguments in conjunction with optional parameters, you can easily omit parameters. Any optional parameters will receive their default value. However, if you omit any mandatory parameters, your code will not compile.

You can mix positional and named arguments. However, you must specify all positional arguments before any named arguments.