# Module 03(Working with methods and Handling Exception)

## 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, andstring 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 methodReturnMultiOut 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 toGetServiceName()

```
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 ref keyword 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.

**Exceptions**

Traditionally, applications used the concept of a global error object. When a piece of code caused an error, it would set the data in this object to indicate the cause of the error and then return to the caller. It was the responsibility of the calling code to examine the error object and determine how to handle it. However, this approach is not robust, because it is too easy for a programmer to forget to handle errors appropriately.

The .NET Framework uses exceptions to help overcome these issues. An exception is an indication of an error or exceptional condition. A method can throw an exception when it detects that something unexpected has happened, for example, the application tries to open a file that does not exist.

**Propagation**

When a method throws an exception, the calling code must be prepared to detect and handle this exception. If the calling code does not detect the exception, the code is aborted and the exception is automatically propagated to the code that invoked the calling code. This process continues until a section of code takes responsibility for handling the exception. Execution continues in this section of code after the exception-handling logic has completed. If no code handles the exception, then the process will terminate and display a message to the user.

**Handling Exceptions**

The try/catch block is the key programming construct that enables you to implement structured exception handling in your applications. You wrap code that may fail and cause an exception in a try block, and add one or more catch blocks to handle any exceptions that may occur.  The recommended strategy to follow with catch blocks is to catch more specific exceptions first, and more general exceptions last.  For example, if you expect to run across a an exception around file access, you would catch the FileNotFoundException in the first catch block and then perhaps create a second catch block that would watch for the generic Exception class to catch any other exception besides the FileNotFoundException.

If the FileNotFoundException is triggered, that catch block will have code to handle that exception.  Otherwise, the other catch block(s) will contain code to handle other exceptions, or to throw the exception back up the stack to the calling application.

The following code example shows the syntax for defining a try/catch block.

```
try
{
   // Try block.
}
catch (FileNotFoundException fnfEx)
{
   // Catch block 1.
}
catch (Exception e)
{
   // Catch block n.
}
```

**Using Finally**

Some methods may contain critical code that must always be run, even if an unhandled exception occurs. For example, a method may need to ensure that it closes a file that it was writing to or releases some other resources before it terminates. A finally block enables you to handle this situation.

You specify a finally block after any catch handlers in a try/catch block. It specifies code that must be performed when the block finishes, irrespective of whether any exceptions, handled or unhandled, occur. If an exception is caught and handled, the exception handler in the catch block will run before the finally block.

You can also add a finally block to code that has no catch blocks. In this case, all exceptions are unhandled, but the finally block will always run.

The following code example shows how to implement a try/catch/finally block.

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
finally
{
   // Code that always runs to close files or release resources.
}
```

**Throwing Exceptions**

You can create an instance of an exception class in your code and throw the exception to indicate that an exception has occurred. When you throw an exception, execution of the current block of code terminates and the CLR passes control to the first available exception handler that catches the exception.

To throw an exception, you use the **throw** keyword and specify the exception object to throw.

The following code example shows how to create an instance of the **NullReferenceException** class and then throw the **ex** object.

```
var ex = new NullReferenceException("The 'Name' parameter is null.");
throw ex;
```

A common strategy is for a method or block of code to catch any exceptions and attempt to handle them. If the catch block for an exception cannot resolve the error, it can rethrow the exception to propagate it to the caller.

The following code example shows how to rethrow an exception that has been caught in a catch block.

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Attempt to handle the exception
```

```
   ...
   // If this catch handler cannot resolve the exception,
   // throw it to the calling code
   throw;
}
```

**Module Three Assignment**

In later modules, you will begin to create class files to represent the items we have been using so far. You will create class files for Students, Teachers, Courses, etc. The class files will consist of attributes and methods. To prepare for this, the assignment for this module will ask you to think about some basic methods that you can implement in your code.

This assignment requires you to create some methods for getting data for your variables and sending that data to the console window.

In the assignment, you are to practice getting values from a user and assigning the to local variables. As a result, move the variables into the appropriate methods. For example, you could create a method called GetStudentInformation() and in that method, you could prompt the user for each piece of student information and then assign it to the variables you created.

Next, create methods such as PrintStudentDetails(string first, string last, string birthday) that accepts the proper variables, and use an appropriate message to print the content to the console window.

The first example is a guide for you, the rest you will need to create on your own.

Create a method that prompts a user of your console application to input the information for a student:

```
static void GetStudentInformation()
{
     Console.WriteLine("Enter the student's first name: ");
     string firstName = Console.ReadLine();
     Console.WriteLine("Enter the student's last name");
     string lastName = Console.ReadLine();
      // Code to finish getting the rest of the student data
     .....
}

static void PrintStudentDetails(string first, string last, string birthday)
{
    Console.WriteLine("{0} {1} was born on: {2}", first, last, birthday);
}
```

1. Using the above partial code sample, complete the method for getting student data.
2. Create a method to get information for a teacher, a course, a uprogram, and a degree using a similar method as above.
3. Create methods to print the information to the screen for each object such as static voidPrintStudentDetails(...).
4. From within Main(), call each of the methods to prompt for input from a user of your application.
5. Just enter enough information to show you understand how to use methods. (At least three attributes each).
6. Assign the values that are input, to the proper variables.
7. Output the values of each object using the "print" methods that you created.

**Exceptions**

1. At times, developers create method signatures early on in the development process but leave the implementation until later. This can lead to methods that are not complete if a developer forgets about these empty methods. One way to help overcome the issue of not remembering to complete a method is to throw an exception in that method if no implementation details are present.
2. For this task, use MSDN to research the NotImplementedException exception.
3. Create a new method for validating a student's birthday. You won't write any validation code in this method, but you will throw the NotImplementedException in this method
4. Call the method from Main() to verify your exception is thrown

**Challenge** (This challenge is for your own study and does not need to be submitted for peer review)

Using MSDN, research the System.DateTime type. Using the information you learn, modify your birth date field for the student and/or teacher to ensure it used a DateTime type if you did not already include that in your data for these objects.

- Remove your NotImplementedException statement in the validate method you created above.

- Create a try/catch block to catch invalid date entries and display a message to the user if this occurs. (Console output)
- *Hint: Look at methods for DateTime to determine ways to check if a valid date is entered.*