Language Preliminaries Contd.

File IO

In the .NET Framework, the System.IO namespace is the region of the base class libraries devoted to file-based (and memory-based) input and output (I/O) services. Like any namespace, System.IO defines a set of classes, interfaces, enumerations, structures, and delegates.

A file is a collection of data stored in a disk with a specific name, a extension and a directory path. When a file is opened for reading or writing purpose, it becomes a stream. The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the input stream and the output stream. The input stream is used for reading data from file (read operation) and the output stream is used for writing into the file (write operation).

File and Directory Operations

The System.IO namespace provides a set of types for performing file and directory operations, such as copying and moving, creating directories, and setting file attributes and permissions. For most features, you can choose between either of two classes, one offering static methods and the other instance methods:

Static classes:

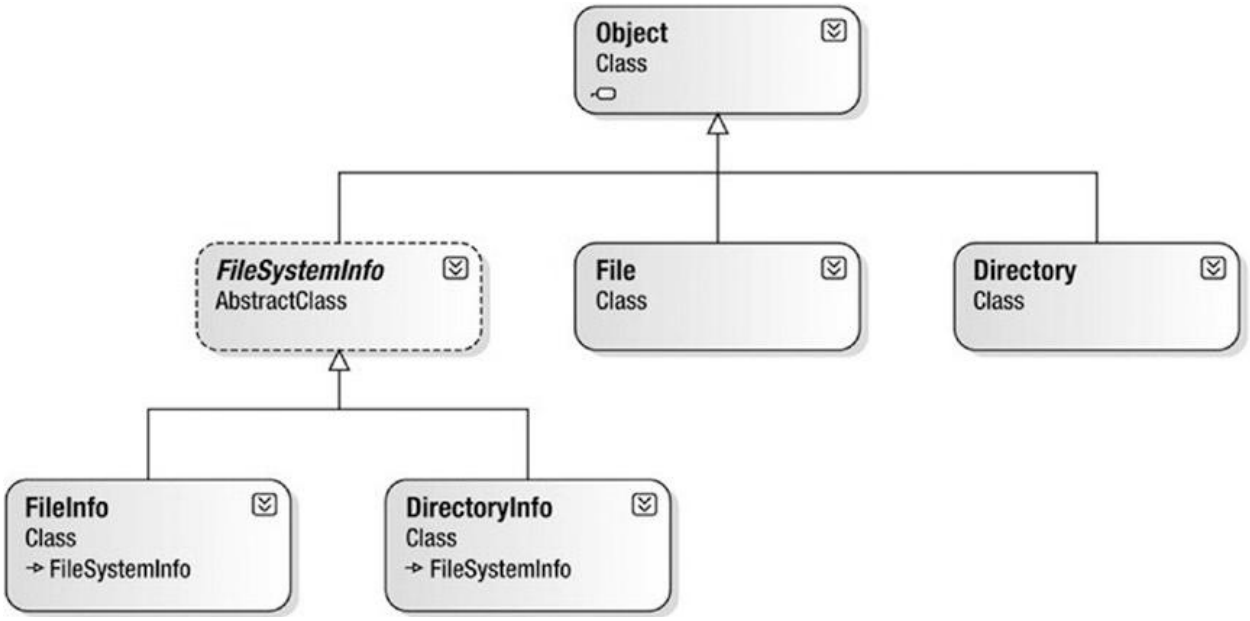
File and Directory

Instance-method classes:

FileInfo and DirectoryInfo

Additionally, there's a static class called Path. This does nothing to files or directories; instead, it provides string manipulation methods for filenames and directory paths. Path also assists with temporary files.

The Directory and File classes directly extend System.Object, Directory and File, expose creation, deletion, copying, and moving operations using various static members. while DirectoryInfo and FileInfo derive from the abstract FileSystemInfo type. The FileInfo and DirectoryInfo types receive many behaviors from the abstract FileSystemInfo base class.



DirectoryInfo :

This class contains a set of members used for creating, moving, deleting, and enumerating over directories and subdirectories. The DirectoryInfo class does not have static methods and can only be used on instantiated objects. The DirectoryInfo object represents a physical directory on a disk.

Key members:

Create(), CreateSubdirectory(), Delete(), GetDirectories(), GetFiles(), MoveTo(), Parent, Root

Example: obtain access to the current working directory using dot (.) notation.

```
DirectoryInfo currentDirectory = new DirectoryInfo(".");
```

Similarly, Windows directory

```
DirectoryInfo windowDirectory = new DirectoryInfo(@"C:\Windows");
```

If you attempt to interact with a nonexistent directory then it will throw an exception. So, you need to create a directory before proceeding as shown below:

```
DirectoryInfo myDirectory = new DirectoryInfo(@"C:\MyDir");
myDirectory.Create();

//Further you can get the detail
```

```
Console.WriteLine("FullName: {0}", myDirectory.FullName);
Console.WriteLine("Name: {0}", myDirectory.Name);
Console.WriteLine("Parent: {0}", myDirectory.Parent);
Console.WriteLine("Creation: {0}", myDirectory.CreationTime);
Console.WriteLine("Attributes: {0}", myDirectory.Attributes);
Console.WriteLine("Root: {0}", myDirectory.Root);
```

//Creating subdirectory

```
myDirectory.CreateSubdirectory(@"MySubDir\MySubdir2");

//Get All directories
DirectoryInfo[] dirList = myDirectory.GetDirectories();

//Delete a directory
DirectoryInfo subDirectory =
           new DirectoryInfo(@"C:\MyDir\MySubDir\MySubDir2");
 subDirectory.Delete(); //MySubDir2 will be deleted
```

FileInfo:

The FileInfo class allows you to obtain details regarding existing files on your hard drive (e.g., time created, size, and file attributes) and supports in the creation, copying, moving, and destruction of files.

Key Members:
Create(), Open(), CopyTo(), MoveTo(), AppendText(), CreateText(), Delete(), Directory, DirectoryName

FileInfo class return a specific I/O-centric object, FileStream that allows you to begin reading and writing data.

```
    public class Program
    {
        static void Main(string[] args)
        {
            FileInfo fi = new FileInfo(@"C:\MyDir\TestFile.txt");
           //assuming MyDir folder exists
            FileStream fs = fi.Create();
           //closing the file stream
           fs.Close();
        }
    }
```

//To delete an existing file

```
FileInfo fi = new FileInfo(@"C:\MyDir\TestFile.txt");
fi.Delete();
```

File

The File type uses several static members to provide functionality almost identical to that of the FileInfo type. Like FileInfo, File supplies AppendText(), Create(), CreateText(), Open(), OpenRead(), OpenWrite(), and OpenText() methods.In many cases, you can use the File and FileInfo types interchangeably.

The following sample code demonstrates how to use the File type to check whether a file exists, and depending on the result, either create a new file and write to it, or open the existing file and read from it.

```
public class Program
{
    static void Main(string[] args)
    {
        string path = @"C:\MyDir\TestFile.txt";
        if (File.Exists(path))
        {
            Console.WriteLine("File exists");
        }
        else
        {
            Console.WriteLine("File doesn't exists");
        }
        // Create a file to write to.
        // Create an instance of StreamWriter to write on thefile.
        // The using statement also closes the StreamReader.
        using (StreamWriter sw = File.CreateText(path))
        {
            sw.WriteLine("First Line");
            sw.WriteLine("Second Line");
            sw.WriteLine("Third Line");
        }

       //appends text to the existing file or create new file
        using (StreamWriter sw = File.AppendText(path))
        {
            sw.WriteLine("Forth Line");
            sw.WriteLine("Fifth Line");
        }
```

```
            // Open the file to read from.
            using (StreamReader sr = File.OpenText(path))
            {
                string s;
                while ((s = sr.ReadLine()) != null)
                {
                    Console.WriteLine(s);
                }
            }
        }
    }
```

Note: using statement defines a boundary for the object outside of which, the object is automatically destroyed.

Directory

The static Directory class provides a set of methods analogous to those in the File class—for checking whether a directory exists (Exists), moving a directory (Move), deleting a directory (Delete), getting/setting times of creation or last access, and getting/setting security permissions.

```
        static void Main(string[] args)
        {
            //get current directory
            string curDir = Directory.GetCurrentDirectory();
            Console.WriteLine(curDir);

            //get root of current directory
            string rootDir = Directory.GetDirectoryRoot(curDir);
            Console.WriteLine(rootDir);

            string path = @"C:\MyDir";
            if (Directory.Exists(path)) //returns true or false
            {
                Console.WriteLine("Directory already exists");
            }
            else
            {
                //creates MyDir directory
                Directory.CreateDirectory(path);
                Console.WriteLine("Directory created");
            }
            // deletes MyDir directory, if it is empty.
            Directory.Delete(path);
```

```
        // The second parameter specifies whether you
        // wish to destroy any subdirectories.
        Directory.Delete(@"C:\MyDir2", false);
    }
```

Try Statements and Exceptions

An exception is a runtime error in a program that violates a system or application constraint, or a condition that is not expected to occur during normal execution of the program. Possible exceptions include attempting to connect to a database that no longer exists when a program tries to divide a number by zero or opening a corrupted XML file. When these occur, the system catches the error and raises an exception.

Syntax:

```
try
{
    // Code to try.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception.
}
finally
{
     // Code to execute after the try (and possibly catch)
}
```

A try statement specifies a code block subject to error-handling or cleanup code. The try block must be followed by one or more catch blocks, a finally block, or both. The catch block executes when an error is thrown in the try block. The finally block executes after execution leaves the try block (or if present, the catch block), to perform cleanup code, regardless of whether an exception was thrown.

A catch block has access to an Exception object that contains information about the error. You use a catch block to either compensate for the error or rethrow the exception. You rethrow an exception if you merely want to log the problem or if you want to rethrow a new, higher-level exception type.

Sample code:

```
    static void Main(string[] args)
    {
        int x = 0;
```

```
        try
        {
            int y = 10 / x;
            Console.WriteLine(y);

            if (y < 1)
                throw new ArithmeticException();

        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("value of x cannot be zero");
        }
        catch (ArithmeticException ex)
        {
            Console.WriteLine("Quotient is less than 1");
        }
        // IOException, ArgumentException, NullReferenceException
        //These are the other type of Exception
        finally
        {
            Console.WriteLine("finally block executed");
        }
    }
```

A finally block always executes, regardless of whether an exception is thrown.Some resource cleanup, It's useful for cleanup tasks such as closing network connections, closing a file, needs to be done even if an exception is thrown.

Exceptions can be thrown either by the runtime or in user code. It might rethrow the same exception, or it might throw a different one. If it throws a different one, it may want to embed the original exception inside the new one so that the calling method can understand the exception history. The InnerException property of the new exception retrieves the original exception.

LINQ (Language Integrated Query)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. It is a set of language and framework features for writing structured type-safe queries over local object collections and remote data sources. LINQ provides a single querying model that can operate against different data domains individually or all together in a single query.

With LINQ, a query is a first-class language construct, just like classes, methods, events. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML)

There are two ways to write queries in LINQ.

Lamda Expression:
Lambda expression is a function without a name. It makes the syntax short and clear. Its scope is limited to where it is used as an expression.

Syntax :

(Input Parameter) => Method Expression

The name of the parameter can be anything and ahead of this parameter (=>) is an Equal to (=) followed by a greater than (>) symbol which is used to send or pass the input parameter from left to right side and on the right-hand side the operation is performed using the input parameter passed from the left-hand side parameters. This whole syntax forms a Lambda Expression.

Example:

```
j  => j + 10
```

Here j is an input parameter followed by => operator and next to the operator there is an expression that adds numeric 10 to the input variable (j). So the output would increment the numeric 10 to the j variable which was the input parameter on the left-hand side of the expression.

Sample Code:

```
string[] words = { "first", "second", "third", "forth", "fifth" };
var retWords = words.Where(w => w.Length <= 5);
```

Query Syntax:

Query syntax is similar to SQL (Structured Query Language) for the database. It is defined within the C# code. To write a LINQ query, you need to follow the syntax hierarchy like as mentioned below.

```
from <variable> in <collection>
<where, joining, grouping, operators etc.> <lambda expression>
<select or groupBy operator> <format the results>
```

Example:

```
var output = from w in words where w.Length <= 5 select w;
```

Sample Code:

```
using System.Linq; //must import

static void Main(string[] args)
{
    int[] nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    var ret = from x in nums where x <= 5 orderby x select x;
    foreach (int i in ret)
    {
        Console.WriteLine(i);
    }
}
```

The following example demonstrates a simple LINQ query that gets all strings from an array which contains 'a'.

```
static void Main(string[] args)
{
  // Data source
  string[] strArr = { "first", "second", "third", "forth" };

  // LINQ Query
  var linqQuery = from name in strArr
                  where name.Contains('i')
                  select name;

  // Query execution
  foreach (var name in linqQuery)
  {
      Console.WriteLine(name);
  }

}
```

Asynchronous Programming

The principle of asynchronous programming is that you write long running function asynchronously.When you execute something asynchronously, you can move on to another task before it finishes. When you execute something synchronously, you wait for it to finish before moving on to another task.

The async/await pattern is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function The async keyword of C# is used to qualify that a method, lambda expression, or anonymous method should be called in an asynchronous manner automatically.

Simply by marking a method with the **async** modifier, the CLR will create a new thread of execution to handle the task at hand. Furthermore, when you are calling an async method, the **await** keyword will automatically pause the current thread from any further activity until the task is complete, leaving the calling thread free to continue on its way.

The **async** and **await** keywords let you write asynchronous code that has the same structure and simplicity as synchronous code while eliminating the plumbing of asynchronous programming. The core of async programming is the **Task** and **Task<T**> objects, which model asynchronous operations. They are supported by the async and await keywords.

The Task is useful to perform the operations asynchronously, the Task class will represent a single operation and that will be executed asynchronously on a thread pool thread rather than synchronously on the main application thread. You need to import System.Threading.Tasks namespace in our program.

Task Example:

```
using System.Threading.Tasks;
public class Program
{
    static void  Main(string[] args)
    {
        Task t1 = new Task(PrintInfo);
        t1.Start();
        Console.WriteLine("Main Thread Completed");
        Console.ReadLine();
    }
    static void PrintInfo()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine("i value: {0}", i);
        }
```

```
            Console.WriteLine("Child Thread Completed");
        }

    }
```

Async Await Example:

```
public class Program
    {
        static void  Main(string[] args)
        {
            TestMethodAsync();
            Console.ReadLine();
        }
        public static async Task TestMethodAsync()
        {
            Task<string> pocessTask = Process();
            // independent work which doesn't need the result of long
Process

            Console.WriteLine("executed");

            //and now call await on the task
            string result = await pocessTask;
            //use the result
            Console.WriteLine(result);
        }

        public static async Task<string>  Process() //  returns an
string from this long running operation
        {
            await Task.Delay(2000); // 2 second delay
            return "Process complete";
        }

    }
```

Attributes

Attributes provide a powerful method of associating metadata, or declarative information, with code , assemblies, types, methods, properties, and so forth. An attribute is a declarative tag that is used to convey information to runtime about the behaviors of various elements.

An attribute can be specified in the code by placing the name of the attribute enclosed in square brackets ([]) above the declaration of the entity to which it applies.

Example:

```
[attribute]
public class SampleClass // Or method
{
    // Code
}
```

Sample Code:

```
public class Program
{
    static void Main(string[] args)
    {
        Print();
    }

    [Obsolete("you should use PrintNew()")]
    static void Print()
    {
        Console.WriteLine("old print method");
    }
    static void PrintNew()
    {
        Console.WriteLine("new print method");
    }
}
```

Attribute classes

A class that derives from the abstract class System.Attribute, whether directly or indirectly, is an attribute class. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration. By convention, attribute classes are named with a suffix of Attribute. Uses of an attribute may either include or omit this suffix.

Attribute usage

The attribute AttributeUsage is used to describe how an attribute class can be used. AttributeUsage has a positional parameter (Positional and named parameters) that enables an attribute class to specify the kinds of declarations on which it can be used.

Example:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class CustomAttribute : Attribute
{
    //code
}
```

Above code defines an attribute class named SimpleAttribute that can be placed on class_declarations and interface_declarations only.

```
[Custom] class ClassName {...}
[Custom] interface InterfaceName {...}
```