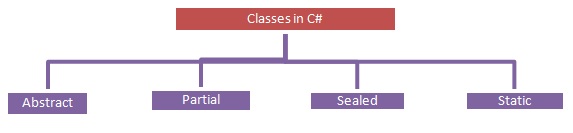
**2. Advance C#**

**2.1 Types class**

* Types of classes in c# :



* Abstract Class :
  + An abstract class is a class that provides a common definition to the subclasses, and this is the type of class whose object is not created.
  + We cannot create an object of an abstract class.
  + It must be inherited in a subclass if you want to use it.
  + An Abstract class contains both abstract and non-abstract methods.
  + The methods inside the abstract class can either have an or no implementation.
  + We can inherit two abstract classes; in this case, implementation of the base class method is optional.
  + An Abstract class has only one subclass.
  + Methods inside the abstract class cannot be private.
  + If there is at least one method abstract in a class, then the class must be abstract.
  + For example

abstract class Accounts

{

}

* Partial Class :
  + It is a type of class that allows dividing their properties, methods, and events into multiple source files, and at compile time, these files are combined into a single class.
  + All the parts of the partial class must be prefixed with the partial keyword.
  + If you seal a specific part of a partial class, the entire class is sealed, the same as for an abstract class.
  + Inheritance cannot be applied to partial classes.
  + The classes written in two class files are combined at run time.
  + For example

partial class Accounts

{

}

* Sealed Class :
  + A Sealed class is a class that cannot be inherited and used to restrict the properties.
  + A Sealed class is created using the sealed keyword.
  + Access modifiers are not applied to a sealed class.
  + To access the sealed members, we must create an object of the class.
  + For example

sealed class Accounts

{

}

* Static Class :
  + It is the type of class that cannot be instantiated. In other words, we cannot create an object of that class using the new keyword, such that class members can be called directly using their name.
  + It was created using the static keyword.
  + Only static members are allowed; in other words, everything inside the class must be static.
  + We cannot create an object of the static class.
  + A Static class cannot be inherited.
  + It allows only a static constructor to be declared.
  + The static class methods can be called using the class name without creating the instance.
  + For example

static class Accounts

{

}

**2.2 Generics**

* Generics in C# provide a way to create classes, interfaces, and methods with placeholders for the data types they work with. This allows you to write code that can work with any data type, providing flexibility and type safety.
* Generics are extensively used in collections (such as List, Dictionary, etc.) and other scenarios where a common functionality is needed for different types.

Generic Classes:

* Syntax :

public class MyClass<T>

{

private T value;

public MyClass(T val)

{

value = val;

}

public T GetValue()

{

return value;

}

}

Generic Methods :

* You can create generic methods inside non-generic classes as follows:

public class MyUtility

{

public T Add<T>(T a, T b)

{

dynamic dynamicA = a;

dynamic dynamicB = b;

return dynamicA + dynamicB;

}

}

Generic Interfaces :

* Syntax :

public interface IRepository<T>

{

void Add(T item);

T GetById(int id);

}

Constraints :

* You can use constraints to specify requirements on the generic type:

public class MyClass<T> where T : IComparable { // Code here }

* This example ensures that T must implement the IComparable interface.

Covariance and Contravariance :

* Generics in C# support covariance and contravariance, allowing more flexibility when working with generic types. This is achieved using the out and in keywords.
* Syntax :

// Covariant interface

public interface IMyCovariant<out T>

{

T GetItem();

}

// Contravariant interface

public interface IMyContravariant<in T>

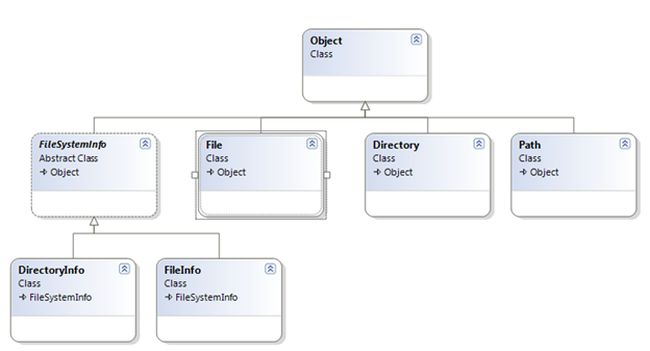
{

void SetItem(T item);

}

**2.3 File system in Depth**

* The System.IO namespace provides four classes that allow you to manipulate individual files, as well as interact with a machine directory structure. The Directory and File directly extends System.Object and supports the creation, copying, moving and deletion of files using various static methods. They only contain static methods and are never instantiated. The FileInfo and DirecotryInfo types are derived from the abstract class FileSystemInfo type and they are typically, employed for obtaining the full details of a file or directory because their members tend to return strongly typed objects. They implement roughly the same public methods as a Directory and a File but they are stateful and the members of these classes are not static.



* The following table outlines the core members of this namespace,

|  |  |
| --- | --- |
| Class Types | Description |
| Directory/ DirectoryInfo | These classes support the manipulation of the system directory structure. |
| DriveInfo | This class provides detailed information regarding the drives that a given machine has. |
| FileStream | This gets you random file access with data represented as a stream of bytes. |
| File/FileInfo | These sets of classes manipulate a computer's files. |
| Path | It performs operations on System.String types that contain file or directory path information in a platform-neutral manner. |
| BinaryReader/ BinaryWriter | These classes allow you to store and retrieve primitive data types as binary values. |
| StreamReader/StreamWriter | Used to store textual information to a file. |
| StringReader/StringWriter | These classes also work with textual information. However, the underlying storage is a string buffer rather than a physical file. |
| BufferedStream | This class provides temp storage for a stream of bytes that you can commit to storage at a later time. |

* The System.IO provides a class DriveInfo to manipulate the system drive related tasks. The DriveInfo class provides numerous details such as the total number of drives, calculation of total hard disk space, available space, drive name, ready status, types and so on.
* The following code snippets perform the rest of the DriveInfo class method operations
* The .NET framework provides the two rudimentary classes, DirectoryInfo and Directory, to do directory-related operations such as creation and deletion.  
  DirectoryInfo Class :
  + The DirectoryInfo class contains a set of members for the creation, deletion, moving and enumeration over directories and subdirectories. Here, in the following code sample, display the information related to temp directory.

DirectoryInfo di=new DirectoryInfo(@"D:\temp");

Console.WriteLine("\*\*\*\*\*\*\*Direcotry Informations\*\*\*\*\*\*\*\n\n");

Console.WriteLine("Full Name={0}",di.FullName);

Console.WriteLine("Root={0}",di.Root);

Console.WriteLine("Attributes={0}", di.Attributes);

Console.WriteLine("Creation Time={0}", di.CreationTime);

Console.WriteLine("Name={0}", di.Name);

Console.WriteLine("Parent={0}", di.Parent);

* + Output :   
    

Directory Class :

* + The Directory class provides nearly the same functionality as DirecotryInfo. The Directory class typically returns string data rather than strongly typed DirectoryInfo objects. The following sample deletes the directory and subdirectory in the D drive.

static void Main(string[] args)

{

     DirectoryInfo di = new DirectoryInfo(@"d:\abc");

     Console.WriteLine("Name:{0}",di.FullName);

     Console.Write("Are you sure to Delete:");

     string str=Console.ReadLine();

     if (str == "y")

     {

         Directory.Delete(@"d:\abc", true);

     }

     Console.Write("Deleted.....");

}

Reading and Writing to Files

* + Reading and writing operations are done using a File object. The following code snippet reads a text file located in the machine somewhere.

private void button1\_Click(object sender, EventArgs e)

{

    try

    {

        textBox2.Text = File.ReadAllText(txtPath.Text);

    }

    catch (FileNotFoundException)

    {

        MessageBox.Show("File not Found....");

    }

}

* + Besides reading a file, we can write some contents over an existing text file by the File class WriteAllTest() method as in the following:
  + File.WriteAllText(@"d:\test.txt", textBox2.Text);
  + It takes a path to save the file and content input method medium such as a text box or any other control.

Stream

* + The .NET provides many objects such as FileStream, StreamReader/Writer, BinaryReader/Writer to read from and write data to a file. A stream basically represents a chunk of data flowing between a source and a destination. Stream provides a common way to interact with a sequence of bytes regardless of what kind of devices store or display the bytes. The following table provides common stream member functions:

|  |  |
| --- | --- |
| Methods | Description |
| Read()/ ReadByte() | Read a sequence of bytes from the current stream. |
| Write()/WriteByte() | Write a sequence of bytes to the current stream. |
| Seek() | Sets the position in the current stream. |
| Position() | Determine the current position in the current stream. |
| Length() | Return the length of the stream in bytes. |
| Flush() | Updates the underlying data source with the current state of the buffer and then clears the buffer. |
| Close() | Closes the current stream and releases any associated stream resources. |

FileStream

* + A FileStream instance is used to read or write data to or from a file. In order to construct a FileStream, first we need a file that we want to access. Second, the mode that indicates how we want to open the file. Third, the access that indicates how we want to access a file. And finally, the share access that specifies whether you want exclusive access to the file.

|  |  |
| --- | --- |
| Enumeration | Values |
| FileMode | Create, Append, Open, CreateNew, Truncate, OpenOrCreate |
| FileAccess | Read, Write, ReadWrite |
| FileShare | Inheritable, Read, None, Write, ReadWrite |

* + The FileStream can read or write only a single byte or an array of bytes. You will be required to encode the System.String type into a corresponding byte array. The System.Text namespace defines a type named encoding that provides members that encode and decode strings to an array of bytes. Once encoded, the byte array is persisted to a file with the FileStream.Write() method. To read the bytes back into memory, you must reset the internal position of the stream and call the ReadByte() method. Finally, you display the raw byte array and the decoded string to the console.

using(FileStream fs=new FileStream(@"d:\ajay123.doc",FileMode.Create))

{

    string msg = "first program";

    byte[] byteArray = Encoding.Default.GetBytes(msg);

    fs.Write(byteArray, 0, byteArray.Length);

    fs.Position = 0;

    byte[] rFile = new byte[byteArray.Length];

    for (int i = 0; i < byteArray.Length; i++)

    {

        rFile[i] = (byte)fs.ReadByte();

        Console.WriteLine(rFile[i]);

    }

    Console.WriteLine(Encoding.Default.GetString(rFile));

}

**2.4 Data Serialization**

* Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be stored and transferred.
* .NET features the following serialization technologies:
  + [JSON serialization](https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/overview) maps .NET objects to and from JavaScript Object Notation (JSON). JSON is an open standard that's commonly used to share data across the web. The JSON serializer serializes public properties by default, and can be configured to serialize private and internal members as well.
  + [XML serialization](https://learn.microsoft.com/en-us/dotnet/standard/serialization/xml-and-soap-serialization) serializes only public properties and fields and does not preserve type fidelity. This is useful when you want to provide or consume data without restricting the application that uses the data. Because XML is an open standard, it is an attractive choice for sharing data across the Web.
  + [Binary serialization](https://learn.microsoft.com/en-us/previous-versions/dotnet/fundamentals/serialization/binary/binary-serialization) preserves type fidelity, which means that the complete state of the object is recorded and when you deserialize, an exact copy is created. This type of serialization is useful for preserving the state of an object between different invocations of an application. For example, you can share an object between different applications by serializing it to the Clipboard. You can serialize an object to a stream, to a disk, to memory, over the network, and so forth. Remoting uses serialization to pass objects "by value" from one computer or application domain to another.