Introduction to .NET Architecture

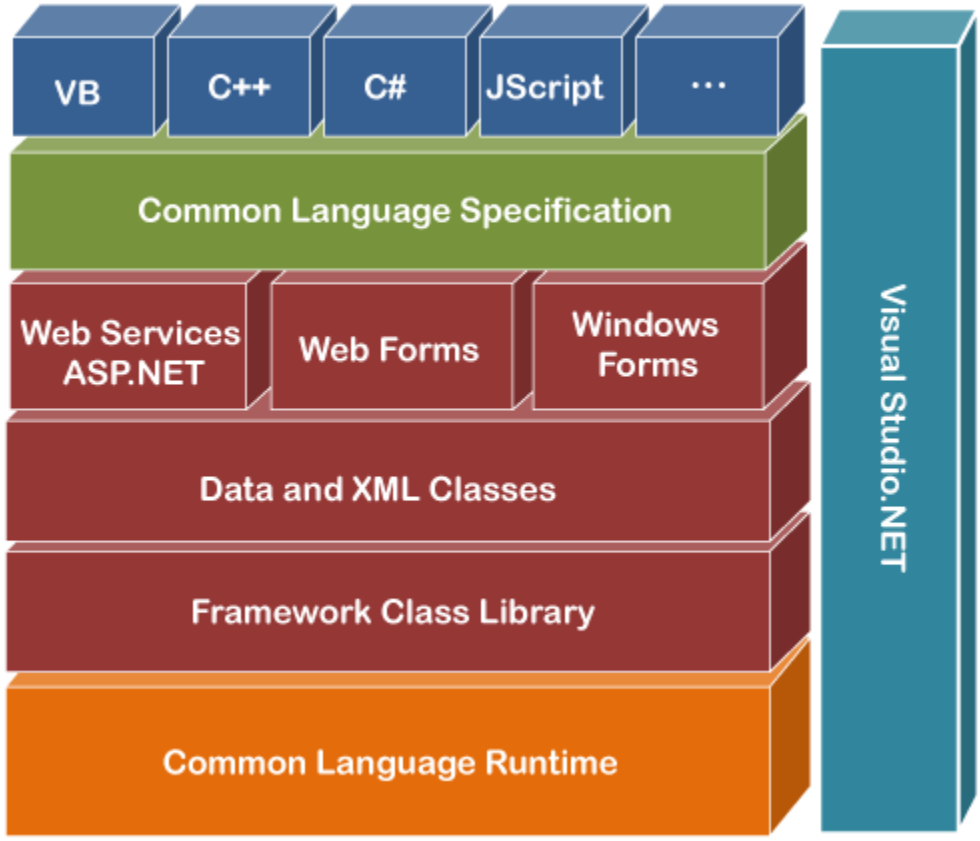
The **.NET Framework** is a software development platform that was introduced by Microsoft in the late 1990 under the NGWS. On 13 February 2002, Microsoft launched the first version of the .NET Framework, referred to as the **.NET Framework 1.0**.

.NET Framework

It is a virtual machine that provide a common platform to run an application that was built using the different language such as C#, VB.NET, Visual Basic, etc. It is also used to create a form based, console-based, mobile and web-based application or services that are available in Microsoft environment. Furthermore, the .NET framework is a pure object oriented, that similar to the Java language. But it is not a platform independent as the Java. So, its application runs only to the windows platform.

The main objective of this framework is to develop an application that can run on the windows platform. The current version of the .Net framework is 4.8.

> *Note: The .NET Framework is not only a language, but it is also a software and language neutral platform.*



Components of .NET Framework

There are following components of .NET Framework:

1. CLR (Common Language Runtime)
2. CTS (Common Type System)
3. BCL (Base Class Library)
4. CLS (Common Language Specification)
5. FCL (Framework Class Library)
6. .NET Assemblies
7. XML Web Services
8. Window Services

CLR (common language runtime)

It is an important part of a .NET framework that works like a virtual component of the .NET Framework to executes the different languages program like c#, Visual Basic, etc. A CLR also helps to convert a source code into the byte code, and this byte code is known as CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language). After converting into a byte code, a CLR uses a JIT compiler at run time that helps to convert a CIL or MSIL code into the machine or native code.

CTS (Common Type System)

It specifies a standard that represent what type of data and value can be defined and managed in computer memory at runtime. A CTS ensures that programming data defined in various languages should be interact with each other to share information. For example, in C# we define data type as int, while in VB.NET we define integer as a data type.

BCL (Base Class Library)

The base class library has a rich collection of libraries features and functions that help to implement many programming languages in the .NET Framework, such as C #, F #, Visual C ++, and more. Furthermore, BCL divides into two parts:

1. **User defined class library**
    - **Assemblies -** It is the collection of small parts of deployment an application's part. It contains either the DLL (Dynamic Link Library) or exe (Executable) file.
        1. In LL, it uses code reusability, whereas in exe it contains only output file/ or application.
        2. DLL file can't be open, whereas exe file can be open.
        3. DLL file can't be run individually, whereas in exe, it can run individually.
        4. In DLL file, there is no main method, whereas exe file has main method.
2. **Predefined class library**

- **Namespace -** It is the collection of predefined class and method that present in .Net. In other languages such as, C we used header files, in java we used package similarly we used "using system" in .NET, where using is a keyword and system is a namespace.

CLS (Common language Specification)

It is a subset of common type system (CTS) that defines a set of rules and regulations which should be followed by every language that comes under the .net framework. In other words, a CLS language should be cross-language integration or interoperability. For example, in C# and VB.NET language, the C# language terminate each statement with semicolon, whereas in VB.NET it is not end with semicolon, and when these statements execute in .NET Framework, it provides a common platform to interact and share information with each other.

Microsoft .NET Assemblies

A .NET assembly is the main building block of the .NET Framework. It is a small unit of code that contains a logical compiled code in the Common Language infrastructure (CLI), which is used for deployment, security and versioning. It defines in two parts (process) DLL and library (exe) assemblies. When the .NET program is compiled, it generates a metadata with Microsoft Intermediate Language, which is stored in a file called Assembly.

FCL (Framework Class Library)

It provides the various system functionality in the .NET Framework, that includes classes, interfaces and data types, etc. to create multiple functions and different types of application such as desktop, web, mobile application, etc. In other words, it can be defined as, it provides a base on which various applications, controls and components are built in .NET Framework.

Key Components of FCL

1. Object type
2. Implementation of data structure

3. Base data types
4. Garbage collection
5. Security and database connectivity
6. Creating common platform for window and web-based application

Characteristics of .NET Framework

1. CLR (Common Language Runtime)
2. Namespace - Predefined class and function
3. Metadata and Assemblies
4. Application domains
5. It helps to configure and deploy the .net application
6. It provides form and web-based services

7. NET and ASP.NET AJAX
8. LINQ
9. Security and Portability
10. Interoperability
11. It provides multiple environments for developing an application

Versions of .NET Framework

1. On 13 February 2002, Microsoft launched first version of .Net framework 1.0.
2. The second version 2.0 of .net framework was launched on 22 January 2006.
3. Third version 3.0 of .Net framework was released on 21 November 2006.
4. A .Net framework version 3.5 was released on 19 November 2007.
5. Version 4.0 of .Net framework was released on 29 September 2008
6. Version 4.5 of .Net framework was released on 15 August 2012.
7. .Net framework 4.5.1 version was announced on 17 October 2013
8. On 5 May 2014, a 4.5.2 version of .Net framework was released.
9. .Net framework 4.6 version was announced on 12 November 2014
10. .Net framework 4.6.1 version was released on 30 October 2015
11. .Net framework 4.6.2 version was announced on March 30, 2016
12. .Net framework 4.7 version was announced on April 5, 2017
13. .Net framework 4.7.1 version was announced on October 17, 2017
14. Version 4.7.2 of .Net framework was released on 30 April 2018.
15. And currently we are using .Net framework version 4.8 that was released on 18 April 2019

Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
```

```
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception

class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

Sr.No.	Exception Class & Description
1	**System.IO.IOException** Handles I/O errors.
2	**System.IndexOutOfRangeException** Handles errors generated when a method refers to an array index out of range.

3	**System.ArrayTypeMismatchException** Handles errors generated when type is mismatched with the array type.
4	**System.NullReferenceException** Handles errors generated from referencing a null object.
5	**System.DivideByZeroException** Handles errors generated from dividing a dividend with zero.
6	**System.InvalidCastException** Handles errors generated during typecasting.
7	**System.OutOfMemoryException** Handles errors generated from insufficient free memory.
8	**System.StackOverflowException** Handles errors generated from stack overflow.

Handling Exceptions

C# provides a structured solution to the exception handling in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs −

```
using System;

namespace ErrorHandlingApplication {
  class DivNumbers {
    int result;

    DivNumbers() {
      result = 0;
    }
    public void division(int num1, int num2) {
      try {
        result = num1 / num2;
      } catch (DivideByZeroException e) {
        Console.WriteLine("Exception caught: {0}", e);
      } finally {
        Console.WriteLine("Result: {0}", result);
      }
    }
    static void Main(string[] args) {
      DivNumbers d = new DivNumbers();
      d.division(25, 0);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

Exception caught: System.DivideByZeroException: Attempted to divide by zero.

Result: 0

Creating User-Defined Exceptions

We can also define your own exception. User-defined exception classes are derived from the **Exception** class. The following example demonstrates this −

```
using System;

namespace UserDefinedException {
  class TestTemperature {
    static void Main(string[] args) {
      Temperature temp = new Temperature();
      try {
        temp.showTemp();
      } catch(TempIsZeroException e) {
        Console.WriteLine("TempIsZeroException: {0}", e.Message);
      }
      Console.ReadKey();
    }
  }
}
public class TempIsZeroException: Exception {
  public TempIsZeroException(string message): base(message) {
  }
}
public class Temperature {
  int temperature = 0;

  public void showTemp() {

    if(temperature == 0) {
      throw (new TempIsZeroException("Zero Temperature found"));
    } else {
      Console.WriteLine("Temperature: {0}", temperature);
```

```
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

TempIsZeroException: Zero Temperature found

Throwing Objects

We can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as −

```
Catch(Exception e) {
  ...
  Throw e
}
```

Multi-Threading

A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.

Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

So far we wrote the programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

Thread Life Cycle

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread −

- **The Unstarted State** − It is the situation when the instance of the thread is created but the Start method is not called.
- **The Ready State** − It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State** − A thread is not executable, when
    - Sleep method has been called
    - Wait method has been called
    - Blocked by I/O operations
- **The Dead State** − It is the situation when the thread completes execution or is aborted.

The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. We can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class MainThreadProgram {
```

```
    static void Main(string[] args) {
      Thread th = Thread.CurrentThread;
      th.Name = "MainThread";

      Console.WriteLine("This is {0}", th.Name);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

This is MainThread

Properties and Methods of the Thread Class

The following table shows some most commonly used **properties** of the **Thread** class −

Sr.No.	Property & Description
1	**CurrentContext** Gets the current context in which the thread is executing.
2	**CurrentCulture** Gets or sets the culture for the current thread.
3	**CurrentPrinciple** Gets or sets the thread's current principal (for role-based security).
4	**CurrentThread**

	Gets the currently running thread.
5	**CurrentUICulture** Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
6	**ExecutionContext** Gets an ExecutionContext object that contains information about the various contexts of the current thread.
7	**IsAlive** Gets a value indicating the execution status of the current thread.
8	**IsBackground** Gets or sets a value indicating whether or not a thread is a background thread.
9	**IsThreadPoolThread** Gets a value indicating whether or not a thread belongs to the managed thread pool.
10	**ManagedThreadId** Gets a unique identifier for the current managed thread.
11	**Name** Gets or sets the name of the thread.

12	**Priority** Gets or sets a value indicating the scheduling priority of a thread.
13	**ThreadState** Gets a value containing the states of the current thread.

The following table shows some of the most commonly used **methods** of the **Thread** class −

Sr.No.	Method & Description
1	**public void Abort()** Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
2	**public static LocalDataStoreSlot AllocateDataSlot()** Allocates an unnamed data slot on all the threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
3	**public static LocalDataStoreSlot AllocateNamedDataSlot(string name)** Allocates a named data slot on all threads. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
4	**public static void BeginCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception might jeopardize other tasks in the application domain.

5	**public static void BeginThreadAffinity()** Notifies a host that managed code is about to execute instructions that depend on the identity of the current physical operating system thread.
6	**public static void EndCriticalRegion()** Notifies a host that execution is about to enter a region of code in which the effects of a thread abort or unhandled exception are limited to the current task.
7	**public static void EndThreadAffinity()** Notifies a host that managed code has finished executing instructions that depend on the identity of the current physical operating system thread.
8	**public static void FreeNamedDataSlot(string name)** Eliminates the association between a name and a slot, for all threads in the process. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
9	**public static Object GetData(LocalDataStoreSlot slot)** Retrieves the value from the specified slot on the current thread, within the current thread's current domain. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
10	**public static AppDomain GetDomain()** Returns the current domain in which the current thread is running.
11	**public static AppDomain GetDomainID()**

	Returns a unique application domain identifier
12	**public static LocalDataStoreSlot GetNamedDataSlot(string name)** Looks up a named data slot. For better performance, use fields that are marked with the ThreadStaticAttribute attribute instead.
13	**public void Interrupt()** Interrupts a thread that is in the WaitSleepJoin thread state.
14	**public void Join()** Blocks the calling thread until a thread terminates, while continuing to perform standard COM and SendMessage pumping. This method has different overloaded forms.
15	**public static void MemoryBarrier()** Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.
16	**public static void ResetAbort()** Cancels an Abort requested for the current thread.
17	**public static void SetData(LocalDataStoreSlot slot, Object data)** Sets the data in the specified slot on the currently running thread, for that thread's current domain. For better performance, use fields marked with the ThreadStaticAttribute attribute instead.

18	**public void Start()** Starts a thread.
19	**public static void Sleep(int millisecondsTimeout)** Makes the thread pause for a period of time.
20	**public static void SpinWait(int iterations)** Causes a thread to wait the number of times defined by the iterations parameter
21	**public static byte VolatileRead(ref byte address)** **public static double VolatileRead(ref double address)** **public static int VolatileRead(ref int address)** **public static Object VolatileRead(ref Object address)** Reads the value of a field. The value is the latest written by any processor in a computer, regardless of the number of processors or the state of processor cache. This method has different overloaded forms. Only some are given above.
22	**public static void VolatileWrite(ref byte address,byte value)** **public static void VolatileWrite(ref double address, double value)** **public static void VolatileWrite(ref int address, int value)** **public static void VolatileWrite(ref Object address, Object value)** Writes a value to a field immediately, so that the value is visible to all processors in the computer. This method has different overloaded forms. Only some are given above.

23	**public static bool Yield()** Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

Creating Threads

Threads are created by extending the Thread class. The extended Thread class then calls the **Start()** method to begin the child thread execution.

The following program demonstrates the concept −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
      Console.WriteLine("Child thread starts");
    }
    static void Main(string[] args) {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");
      Thread childThread = new Thread(childref);
      childThread.Start();
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

In Main: Creating the Child thread
Child thread starts

Managing Threads

The Thread class provides various methods for managing threads.

The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
      Console.WriteLine("Child thread starts");

      // the thread is paused for 5000 milliseconds
      int sleepfor = 5000;

      Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
      Thread.Sleep(sleepfor);
      Console.WriteLine("Child thread resumes");
    }

    static void Main(string[] args) {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");

      Thread childThread = new Thread(childref);
      childThread.Start();
```

```
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

Destroying Threads

The **Abort()** method is used for destroying threads.

The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this −

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class ThreadCreationProgram {
    public static void CallToChildThread() {
      try {
        Console.WriteLine("Child thread starts");

        // do some work, like counting to 10
        for (int counter = 0; counter <= 10; counter++) {
          Thread.Sleep(500);
          Console.WriteLine(counter);
```

```
        }

        Console.WriteLine("Child Thread Completed");
      } catch (ThreadAbortException e) {
        Console.WriteLine("Thread Abort Exception");
      } finally {
        Console.WriteLine("Couldn't catch the Thread Exception");
      }
    }
    static void Main(string[] args) {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");

      Thread childThread = new Thread(childref);
      childThread.Start();

      //stop the main thread for some time
      Thread.Sleep(2000);

      //now abort the child
      Console.WriteLine("In Main: Aborting the Child thread");

      childThread.Abort();
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

In Main: Creating the Child thread

Child thread starts

0

1

2

In Main: Aborting the Child thread

Thread Abort Exception

Couldn't catch the Thread Exception

C# - File I/O

A **file** is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, 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).

C# I/O Classes

The System.IO namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace −

Sr.No.	I/O Class & Description
1	**BinaryReader** Reads primitive data from a binary stream.
2	**BinaryWriter**

	Writes primitive data in binary format.
3	**BufferedStream** A temporary storage for a stream of bytes.
4	**Directory** Helps in manipulating a directory structure.
5	**DirectoryInfo** Used for performing operations on directories.
6	**DriveInfo** Provides information for the drives.
7	**File** Helps in manipulating files.
8	**FileInfo** Used for performing operations on files.
9	**FileStream** Used to read from and write to any location in a file.
10	**MemoryStream** Used for random access to streamed data stored in memory.

11	**Path** Performs operations on path information.
12	**StreamReader** Used for reading characters from a byte stream.
13	**StreamWriter** Is used for writing characters to a stream.
14	**StringReader** Is used for reading from a string buffer.
15	**StringWriter** Is used for writing into a string buffer.

The FileStream Class

The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a **FileStream** object to create a new file or open an existing file. The syntax for creating a **FileStream** object is as follows −

```
FileStream <object_name> = new FileStream( <file_name>, <FileMode Enumerator>,
   <FileAccess Enumerator>, <FileShare Enumerator>);
```

For example, we create a FileStream object **F** for reading a file named **sample.txt as shown** −

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read,
   FileShare.Read);
```

Sr.No.	Parameter & Description
1	**FileMode** The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are − • **Append** − It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. • **Create** − It creates a new file. • **CreateNew** − It specifies to the operating system, that it should create a new file. • **Open** − It opens an existing file. • **OpenOrCreate** − It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. • **Truncate** − It opens an existing file and truncates its size to zero bytes.
2	**FileAccess** **FileAccess** enumerators have members: **Read**, **ReadWrite** and **Write**.
3	**FileShare** **FileShare** enumerators have the following members − • **Inheritable** − It allows a file handle to pass inheritance to the child processes • **None** − It declines sharing of the current file • **Read** − It allows opening the file for readin. • **ReadWrite** − It allows opening the file for reading and writing • **Write** − It allows opening the file for writing

Example

The following program demonstrates use of the **FileStream** class −

```
using System;
using System.IO;

namespace FileIOApplication {
  class Program {
    static void Main(string[] args) {
      FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate,
        FileAccess.ReadWrite);

      for (int i = 1; i <= 20; i++) {
        F.WriteByte((byte)i);
      }
      F.Position = 0;
      for (int i = 0; i <= 20; i++) {
        Console.Write(F.ReadByte() + " ");
      }
      F.Close();
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1

Advanced File Operations in C#

The preceding example provides simple file operations in C#. However, to utilize the immense powers of C# System.IO classes, you need to know the commonly used properties and methods of these classes.

Sr.No.	Topic & Description
1	Reading from and Writing into Text files It involves reading from and writing into text files. The **StreamReader** and **StreamWriter** class helps to accomplish it.
2	Reading from and Writing into Binary files It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** class helps to accomplish this.
3	Manipulating the Windows file system It gives a C# programamer the ability to browse and locate Windows files and directories.

BinaryReader / BinaryWriter

C# BinaryReader class is used to read binary information from stream. It is found in System.IO namespace. It also supports reading string in specific encoding.

```
using System;

using System.IO;

namespace BinaryWriterExample

{

   class Program

   {

      static void Main(string[] args)

      {
```

```
        WriteBinaryFile();
        ReadBinaryFile();
        Console.ReadKey();
    }
    static void WriteBinaryFile()
    {
        using (BinaryWriter writer = new BinaryWriter(File.Open("e:\\binaryfile.dat", FileMode.Create)))
        {

            writer.Write(12.5);
            writer.Write("this is string data");
            writer.Write(true);
        }
    }
    static void ReadBinaryFile()
    {
using (BinaryReader reader = new BinaryReader(File.Open("e:\\binaryfile.dat", FileMode.Open)))
        {
            Console.WriteLine("Double Value : " + reader.ReadDouble());
            Console.WriteLine("String Value : " + reader.ReadString());
```

```
            Console.WriteLine("Boolean Value : " + reader.ReadBoolean());
        }
    }
}
}
```

Output:

Double Value : 12.5

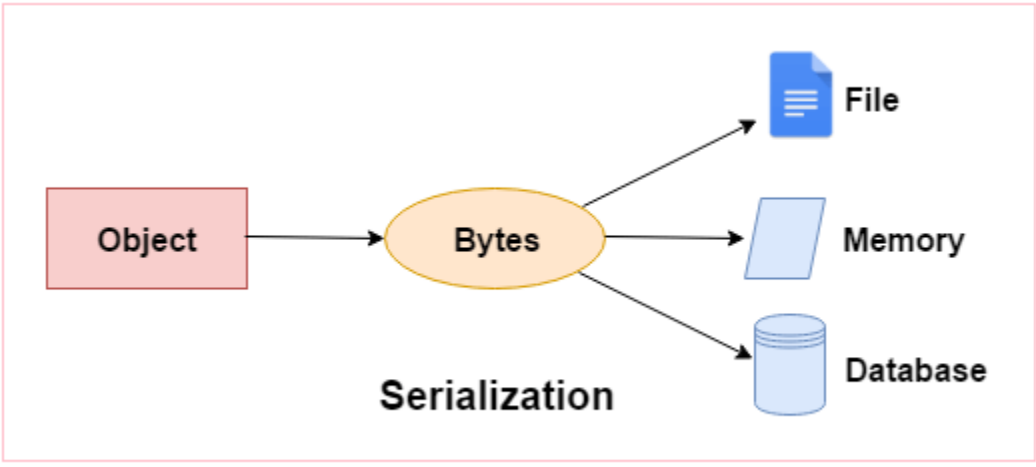
String Value : this is string data

Boolean Value : true

C# Serialization

In C#, serialization is the process of converting object into byte stream so that it can be saved to memory, file or database. The reverse process of serialization is called deserialization.

Serialization is internally used in remote applications.



To serialize the object, we need to apply Serializable Attribute to the type. If we don't apply Serializable attribute to the type, SerializationException exception is thrown at runtime.

Let's see the simple example of serialization in C# where we are serializing the object of Student class. Here, we are going to use BinaryFormatter.Serialize(stream, reference) method to serialize the object.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Student
{
   int rollno;
   string name;
   public Student(int rollno, string name)
   {
      this.rollno = rollno;
      this.name = name;
   }
}
public class SerializeExample
{
```

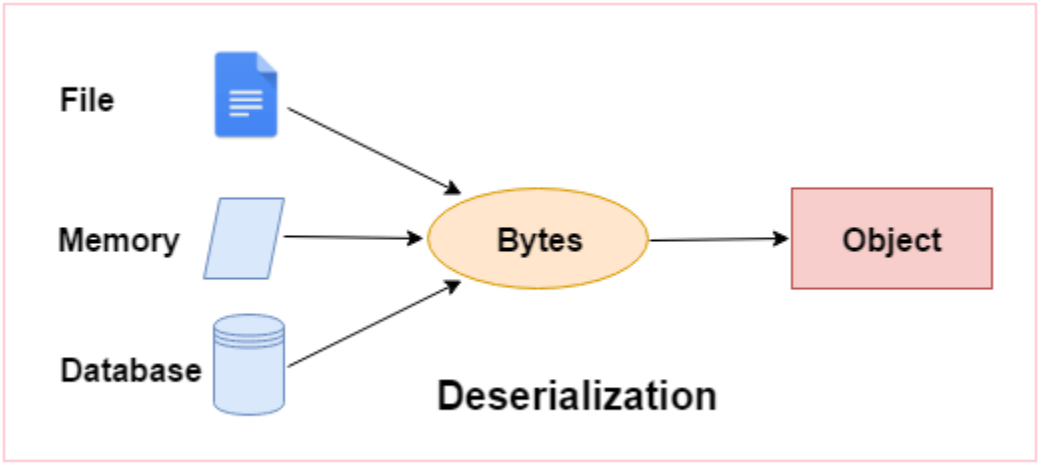
```
    public static void Main(string[] args)
    {
        FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
        BinaryFormatter formatter=new BinaryFormatter();

        Student s = new Student(101, "sonoo");
        formatter.Serialize(stream, s);

        stream.Close();
    }
}
```

C# Deserialization

In C# programming, deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use BinaryFormatter.Deserialize(stream) method to deserialize the stream.



```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Student
{
   public int rollno;
   public string name;
   public Student(int rollno, string name)
   {
      this.rollno = rollno;
      this.name = name;
   }
}
public class DeserializeExample
{
   public static void Main(string[] args)
   {
      FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
      BinaryFormatter formatter=new BinaryFormatter();
```

```
        Student s=(Student)formatter.Deserialize(stream);
        Console.WriteLine("Rollno: " + s.rollno);
        Console.WriteLine("Name: " + s.name);


        stream.Close();
    }
}
```