Kashyap Sayani

Module – 4

○ Enumerations

Enumeration (or enum) is a value data type in C#. It is mainly used to assign the names or string values to integral constants, that make a program easy to read and maintain. For example, the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.). The main objective of enum is to define our own data types(Enumerated Data Types). Enumeration is declared using ***enum*** keyword directly inside a namespace, class, or structure.

Syntax:

```
enum Enum_variable
{
    string_1...;
    string_2...;
   .
   .
}
```

In above syntax, Enum_variable is the name of the enumerator, and string_1 is attached with value 0, string_2 is attached value 1 and so on. Because by default, the first member of an enum has the value 0, and the value of each successive enum member is increased by 1. We can change this default value.

○ Handling Exception

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:

```
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
}
```

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.

○ Basic File Operations

- 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

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>);
```

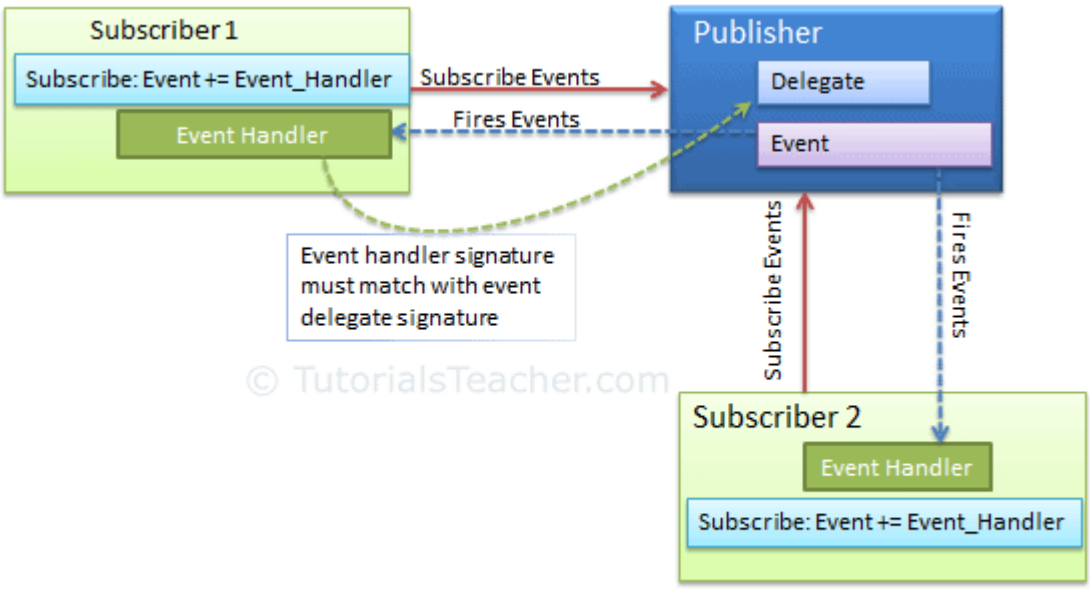
For example, we create a FileStream object **F** for reading a file named **sample.txt as shown −**

```
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

○ Event

- An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.
- The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.
- In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.
- The following figure illustrates the event in C#.



Event Publisher & Subscriber

○ Interface & Inheritance

- Interface

An `interface` is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

Example

```
// interface

interface Animal

{

  void animalSound(); // interface method (does not have a body)

  void run(); // interface method (does not have a body)

}
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default `abstract` and `public`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

- Inheritance

- In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.
- In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

- **Advantage of C# Inheritance**

- **Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.