Module-4

20.Enumeration

- Enumeration 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, natural enumerated types the planets, days of the week, colors, directions, etc.
- The main objective of enumeration is to define our own data types.
- Enumeration is declared using enum keyword directly inside a namespace, class, or structure.

Syntax:

```
enum Enum_variable
{
    string1;
    string2;
    string3;
....
}
```

- By default, the associated constant values of enum members are of type int they start with zero and increase by one following the definition text order.
- You can explicitly specify any other integral numeric type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values.
- You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an extension method.

21.Handling Exceptions

- An exception is defined as an event that occurs during the execution of a program that is unexpected by the program code.
- The actions to be performed in case of occurrence of an exception is not known to the program. In such a case, we create an exception object and call the exception handler code.
- The execution of an exception handler so that the program code does not crash is called exception handling.
- Exception handling is important because it gracefully handles an unwanted event, an exception so that the program code still makes sense to the user.
- Exception handling in C#, supported by the try catch and finally block is a mechanism to detect and handle run-time errors in code.
- The .NET framework provides built-in classes for common exceptions.

try	Used to define a try block. This block holds the code that may throw an exception.
catch	Used to define a catch block. This block catches the exception thrown by the try block.
finally	Used to define the finally block. This block holds the default code.
throw	Used to throw an exception manually.

Syntax:

```
try
{
// Statement which can cause an exception.
}
Catch (Type x)
{
// Statements for handling the exception
}
```

```
finally
{
//Any code
}
```

- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- But in C#, both catch and finally blocks are optional.
- The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.
- If there is no exception occurred inside the try block, the control directly transfers to finally block.
- We can say that the statements inside the finally block is executed always.
- Note that it is an error to transfer control out of a finally block by using break, continue, return or goto.

Catch Blocks

- A catch block can specify the type of exception to catch.
- The type specification is called an exception filter.
- The exception type should be derived from Exception.
- In general, don't specify Exception as the exception filter unless either you know how to handle all exceptions that might be thrown in the try block, or you've included a throw statement at the end of your catch block.
- Multiple catch blocks with different exception classes can be chained together.
- The catch blocks are evaluated from top to bottom in your code, but only one catch block is executed for each exception that is thrown.
- The first catch block that specifies the exact type or a base class of the thrown exception is executed.
- If no catch block specifies a matching exception class, a catch block that doesn't have any type is selected, if one is present in the statement.
- It's important to position catch blocks with the most specific (that is, the most derived) exception classes first.

Finally Blocks

- A finally block enables you to clean up actions that are performed in a try block.
- If present, the finally block executes last, after the try block and any matched catch block.
- A finally block always runs, whether an exception is thrown or a catch block matching the exception type is found.
- The finally block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects.

Standard Exceptions

- There are two types of exceptions: exceptions generated by an executing program and exceptions generated by the common language runtime.
- System.Exception is the base class for all exceptions in C#. Several exception classes inherit from this class including ApplicationException and SystemException.
- These two classes form the basis for most other runtime exceptions.
- Other exceptions that derive directly from System.Exception include IOException, WebException etc.
- The common language runtime throws SystemException.
- The ApplicationException is thrown by a user program rather than the runtime.
- The SystemException includes the ExecutionEngineException, StaclOverFlowException etc.

Exception Classes

1. System.OutOfMemoryException
2. System.NullReferenceException
3. Syste.InvalidCastException
4. Syste.ArrayTypeMismatchException
5. System.IndexOutOfRangeException
6. System.ArithmeticException

7. System.DevideByZeroException
8. System.OverFlowException

22.Events

- Event can be subscriber, publisher, subscriber, notification, and a handler. Generally, the User Interface uses the events.
- C# and .NET support event driven programming via delegates.
- Delegates and events provide notifications to client applications when some state changes of an object.
- It is an encapsulation of idea that Something happened.
- Events and Delegates are tightly coupled concept because event handling requires delegate implementation to dispatch events.
- The class that sends or raises an event is called a Publisher and class that receives or handle the event is called "Subscriber".

Key Points about the Events are:

- In C#, event handler will take the two parameters as input and return the void.
- The first parameter of the Event is also known as the source, which will publish the object.
- The publisher will decide when we have to raise the Event, and the subscriber will determine what response we have to give.
- Event can contain many subscribers.
- Generally, we used the Event for the single user action like clicking on the button.
- If the Event includes the multiple subscribers, then synchronously event handler invoked.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

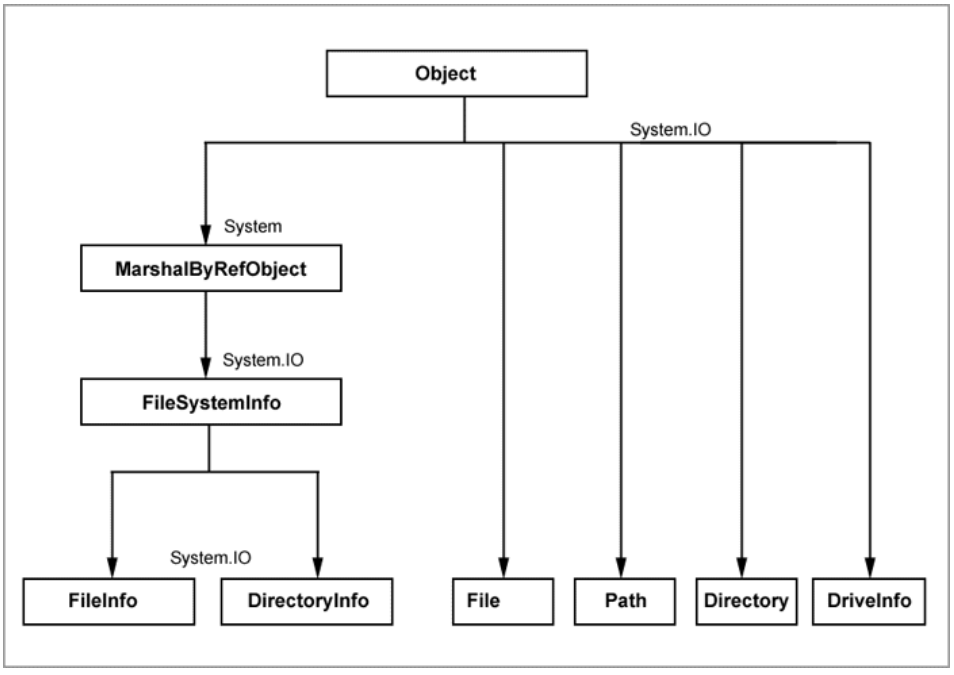
```
public delegate string BoilerLogHandler(string str);
```

then, declare the event using the event keyword

```
event BoilerLogHandler BoilerEventLog;
```

23.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:
    1. the input streams
    2. the output streams
- 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).
- In C#, System.IO namespace contains classes which handle input and output streams and provide information about file and directory structure.



The following table describes some commonly used classes in the System.IO namespace.

Class Name	Description
FileStream	It is used to read from and write to any location within a file.
BinaryReader	It is used to read primitive data types from a binary stream.
BinaryWriter	It is used to write primitive data types in binary format.
StreamReader	It is used to read characters from a byte Stream.
StreamWriter	It is used to write characters to a stream.
StringReader	It is used to read from a string buffer.
StringWriter	It is used to write into a string buffer.
DirectoryInfo	It is used to perform operations on directories.
FileInfo	It is used to perform operations on files.

StreamWriter Class

The StreamWriter class in inherited from the abstract class TextWriter. The TextWriter class represents a writer, which can write a series of characters.

The following table describes some of the methods used by StreamWriter class.

Methods	Description
Close	Closes the current StreamWriter object and the underlying stream.
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
Write	Writes to the stream.
WriteLine	Writes data specified by the overloaded parameters, followed by end of line.

StreamReader Class

The StreamReader class is inherited from the abstract class TextReader. The TextReader class represents a reader, which can read series of characters.

The following table describes some methods of the StreamReader class.

Methods	Description
Close	Closes the object of StreamReader class and the underlying stream, and release any system resources associated with the reader.
Peek	Returns the next available character but doesn't consume it.
Read	Reads the next character or the next set of characters from the stream.
ReadLine	Reads a line of characters from the current stream and returns data as a string.
Seek	Allows the read/write position to be moved to any position with the file.

- **File.AppendText()** is an inbuilt File class method which is used to create a StreamWriter that appends UTF-8 encoded text to an existing file else it creates a new file if the specified file does not exist.
    - **Syntax:**
      public static System.IO.StreamWriter AppendText (string path);

- **File.Delete(String)** is an inbuilt File class method which is used to delete the specified file.
    - **Syntax:**
      public static void Delete (string path);

- **File.Copy(String, String)** is an inbuilt File class method that is used to copy the content of the existing source file content to another destination file which is created by this function.
    - **Syntax:**
      public static void Copy (string sourceFileName, string destFileName)

- **File.Move()** is an inbuilt File class method that is used to move a specified file to a new location. This method also provides the option to specify a new file name.
    - **Syntax:**
      public static void Move (string sourceFileName, string destFileName);

24.Interface & inheritance

Interface

- Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.
- It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.
- Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.
- An interface can be a member of a namespace or a class. An interface declaration can contain declarations (signatures without any implementation) of the following members:
    1. Methods
    2. Properties
    3. Events
- These preceding member declarations typically do not contain a body. Beginning with C# 8.0, an interface member may declare a body. This is called a default implementation. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation.

Syntax for Interface Declaration:

```
interface <interface Name >
{
   // declare Events
   // declare indexers
   // declare methods
   // declare properties
}
```

- Interface cannot include private, protected, or internal members. All the members are public by default.
- Interface cannot contain fields, and auto-implemented properties.
- A class or a struct can implement one or more interfaces implicitly or explicitly. Use public modifier when implementing interface implicitly, whereas don't use it in case of explicit implementation.
- Implement interface explicitly using InterfaceName.MemberName.
- An interface can inherit one or more interfaces.

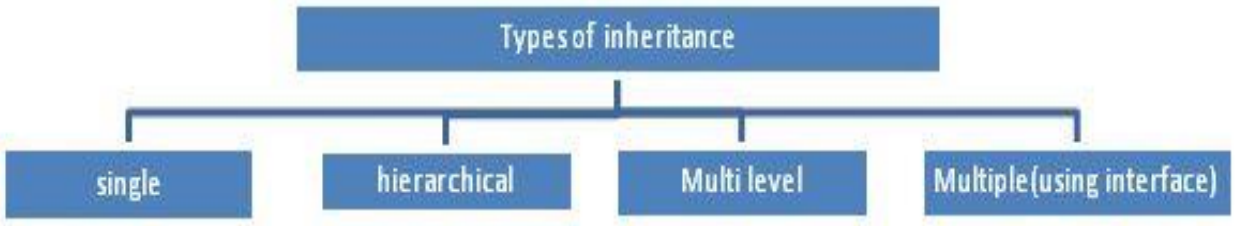
Inheritance

- 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.

Derived Class (child) - the class that inherits from another class

Base Class (parent) - the class being inherited from

To inherit from a class, use the : symbol.



- Single inheritance
    - It is the type of inheritance in which there is one base class and one derived class.
- Hierarchical inheritance
    - This is the type of inheritance in which there are multiple classes derived from one base class.
    - This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.
- Multilevel inheritance
    - When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.
- Multiple inheritance (using Interfaces)
    - C# does not support multiple inheritances of classes.
    - C# does not support multiple inheritances of classes, the same thing can be done using interfaces.
    - Private members are not accessed in a derived class when one class is derived from another.

Syntax:

```
class derived-class : base-class
{
  // methods and fields

}
```