

Module-4

20. Enumeration.

An enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword. C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

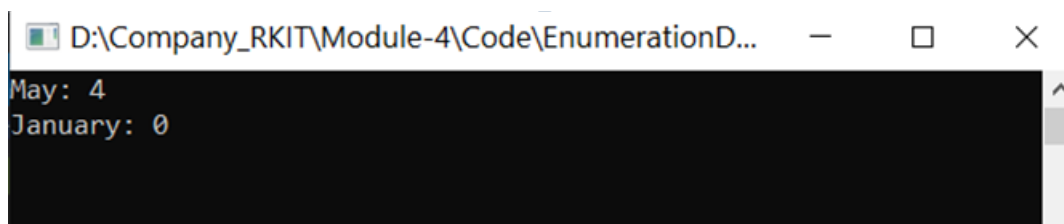
The general syntax for declaring an enumeration is –

```
enum {  
  
enumeration list  
  
};
```

Code:

```
using System;  
  
namespace EnumerationDemo  
{  
    // making an enumerator 'Months'  
    enum Months  
    {  
        // following are the data members  
        January, February, March, April, May, June, July,  
        August, September, October, November, December  
    };  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // getting the integer values of data members  
            int month1 = (int)Months.May;  
            int month2 = (int)Months.January;  
            Console.WriteLine("May: {0}", month1);  
            Console.WriteLine("January: {0}", month2);  
            Console.Read();  
        }  
    }  
}
```

Output:



```
D:\Company_RKIT\Module-4\Code\EnumerationD...  
May: 4  
January: 0
```

21. 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  
}
```

Exception classes:

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.

Code:

```
using System;
```

```
namespace ExceptionHandling
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] arr = { 19, 0, 55, 59 };
```

```
            try
```

```
            {
```

```
                // Try to generate an exception
```

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

```
                {
```

```
                    Console.WriteLine(arr[i] / arr[i + 1]);
```

```
                }
```

```
            }
```

```

// Catch block for invalid array access
catch (IndexOutOfRangeException e)
{

    Console.WriteLine("An Exception has occurred : {0}", e.Message);
}

// Catch block for attempt to divide by zero
catch (DivideByZeroException e)
{

    Console.WriteLine("An Exception has occurred : {0}", e.Message);
}

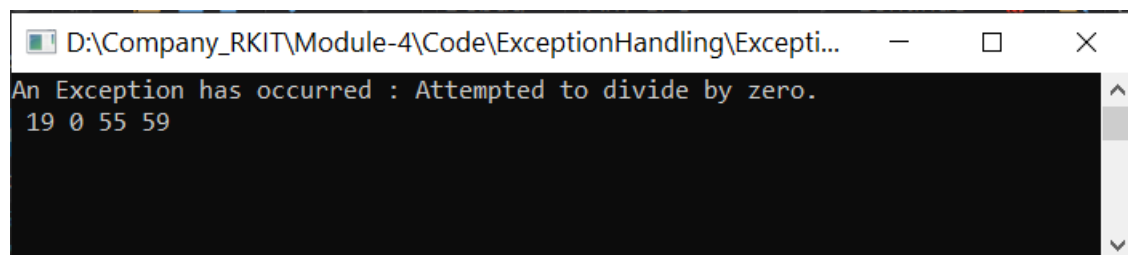
// Catch block for value being out of range
catch (ArgumentOutOfRangeException e)
{

    Console.WriteLine("An Exception has occurred : {0}", e.Message);
}

// Finally block
// Will execute irrespective of the above catch blocks
finally
{
    for (int i = 0; i < arr.Length; i++)
    {
        Console.Write(" {0}", arr[i]);
    }
}
Console.Read();
}
}
}

```

Output:



```

D:\Company_RKIT\Module-4\Code\ExceptionHandling\Excepti...
An Exception has occurred : Attempted to divide by zero.
19 0 55 59

```

22. Events.

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur.

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the publisher class. The classes that receive (or handle) the event are called subscribers. A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime. Delegates are especially used for

implementing events and the call-back methods. All delegates are implicitly derived from the `System.Delegate` class.

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

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

then, declare the event using the event keyword –

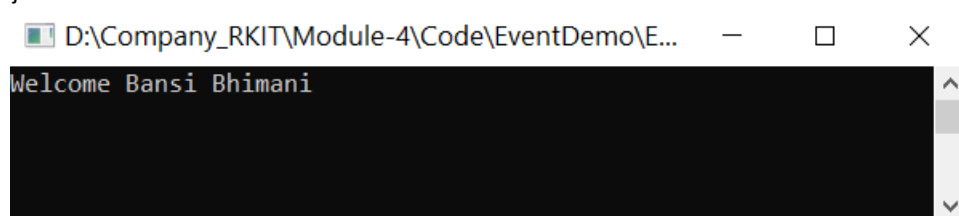
```
event BoilerLogHandler BoilerEventLog;
```

Example of events by delegated:

```
using System;
```

```
namespace EventDemo
```

```
{  
    //declare a 'MyDel' delegate  
    public delegate string MyDel(string str);  
    class EventProgram  
    {  
        //declare the 'MyEvent' event of delagate 'MyDel'  
        event MyDel MyEvent;  
        public EventProgram()  
        {  
            this.MyEvent += new MyDel(this.WelcomeUser);  
        }  
        public string WelcomeUser(string username)  
        {  
            return "Welcome " + username;  
        }  
        static void Main(string[] args)  
        {  
            //creating a object of EventProgram  
            EventProgram objeventprogram = new EventProgram();  
            string result = objeventprogram.MyEvent("Bansi Bhimani");  
            Console.WriteLine(result);  
            Console.Read();  
        }  
    }  
}
```



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: 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).

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> );
```

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.

FileAccess :

FileAccess enumerators have members: Read, ReadWrite and Write.

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 reading.
- ReadWrite – It allows opening the file for reading and writing
- Write – It allows opening the file for writing

Code:

```
using System;
using System.IO;

namespace FileDemo
{
    class Program
    {
        static void Main(string[] args)
        {

            string s = @"D:\Company_RKIT\Module-4\Code\FileDemo\Source\Test.txt";
            string d = @"D:\Company_RKIT\Module-4\Code\FileDemo\Destination\Test.txt";

            //copy and delete operation in file
            try
            {
                File.Copy(s,d, true);
                Console.WriteLine("File copied from Source to Destination...");
                File.Delete(s);
                Console.WriteLine("File deleted from Source...");
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine("File not found exception - {0} , occurred ", e.StackTrace);
            }

            //move operation in file
            try
            {
                File.Move(d, s);
                Console.WriteLine("File moved from Destination to Source again...");
            }
            catch (Exception) { }
            FileStream objfilestream = new FileStream("test.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
            for (int i = 1; i < 10; i++)
            {
                objfilestream.WriteByte((byte)i);
            }
            objfilestream.Position = 0;
            for (int i = 0; i < 10; i++)
            {
                Console.Write(objfilestream.ReadByte() + " ");
            }
            objfilestream.Close();
            Console.Read();

        }
    }
}
```

Output:

```
D:\Company_RKIT\Module-4\Code\FileDemo\Fil...
File copied from Source to Destination...
File deleted from Source...
File moved from Destination to Source again...
1 2 3 4 5 6 7 8 9 -1
```

24. Interface & Inheritance.

Interface:

Like a class, Interface can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members. The implementation of the interface's members will be given by class who implements the interface implicitly or explicitly.

- Interfaces specify what a class must do and not how.
- Interfaces can't have private members.
- By default all the members of Interface are public and abstract.
- The interface will always defined with the help of keyword 'interface'.
- Interface cannot contain fields because they represent a particular implementation of data.
- Multiple inheritance is possible with the help of Interfaces but not with classes.

Syntax for Interface Declaration:

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

Syntax for Implementing Interface:

```
class class_name : interface_name
```

To declare an interface, use *interface* keyword. It is used to provide total abstraction. That means all the members in the interface are declared with the empty body and are public and abstract

by default. A class that implements interface must implement all the methods declared in the interface.

Code:

```
using System;

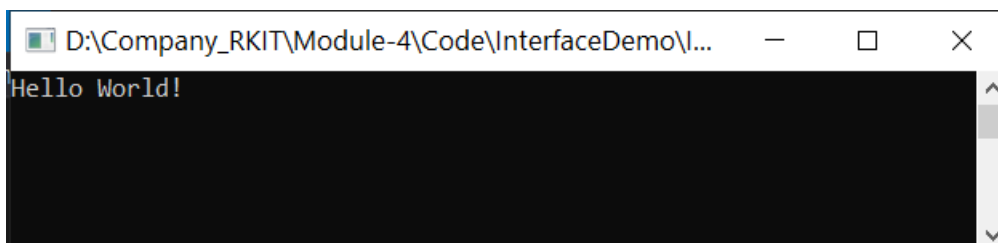
namespace InterfaceDemo
{
    // A simple interface
    interface inter
    {
        // method having only declaration
        // not definition
        void display();
    }

    // A class that implements interface.
    class Program : inter
    {
        // providing the body part of function
        public void display()
        {
            Console.WriteLine("Hello World!");
        }

        // Main Method
        static void Main(string[] args)
        {
            // Creating object
            Program objprogram = new Program();

            // calling method
            objprogram.display();
            Console.Read();
        }
    }
}
```

Output:



Inheritance:

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should

inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class. The idea of inheritance implements the IS-A relationship. For example, mammal IS A animal, dog IS-A mammal hence dog IS-A animal as well, and so on.

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows :

```
<access-specifier>class<base_class> {
```

```
...
```

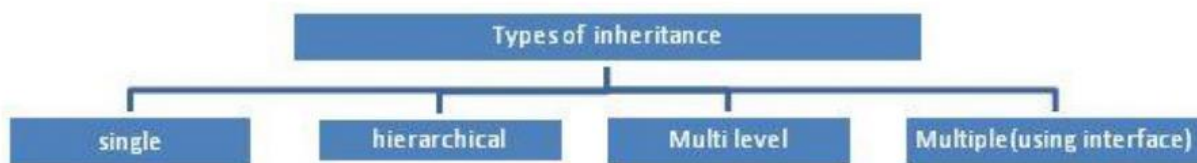
```
}
```

```
class <derived_class>:<base_class> {
```

```
...
```

```
}
```

The following are the types of inheritance in C#.



Single Inheritance:

It is the type of inheritance in which there is one base class and one derived class.

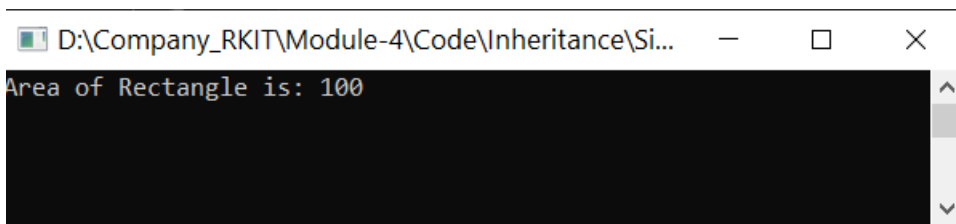
Code:

```
using System;

namespace SingleInheritance
{
    //Base class
    class Shape
    {
        public int width;
        public int height;
        public void setWidth(int w)
        {
            width = w;
        }
        public void setHeight(int h)
        {
            height = h;
        }
    }
}
```

```
// Derived class
class Rectangle : Shape
{
    public int getArea()
    {
        return (width * height);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Rectangle objrectangle = new Rectangle();
        objrectangle.setWidth(10);
        objrectangle.setHeight(10);
        // Print the area of the object.
        Console.WriteLine("Area of Rectangle is: {0}", objrectangle.getArea());
        Console.Read();
    }
}
```

Output:



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.

Code:

```
using System;

namespace HierarchicalInheritance
{
    //base class
    class Father
    {
        public void FatherName()
        {
            Console.WriteLine( "My Father Name is Ramesh");
        }
    }
    //Derived class
    class Son : Father
    {
        public void SonName()
```

```

    {
        Console.WriteLine("My Name is Rakesh");
    }
}
// Derived class
class Daughter : Father
{
    public void DaughterName()
    {
        Console.WriteLine("My Name is Ankita");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Son objson = new Son();

        // Displaying Son Name and Father Name for Son
        objson.SonName();
        objson.FatherName();

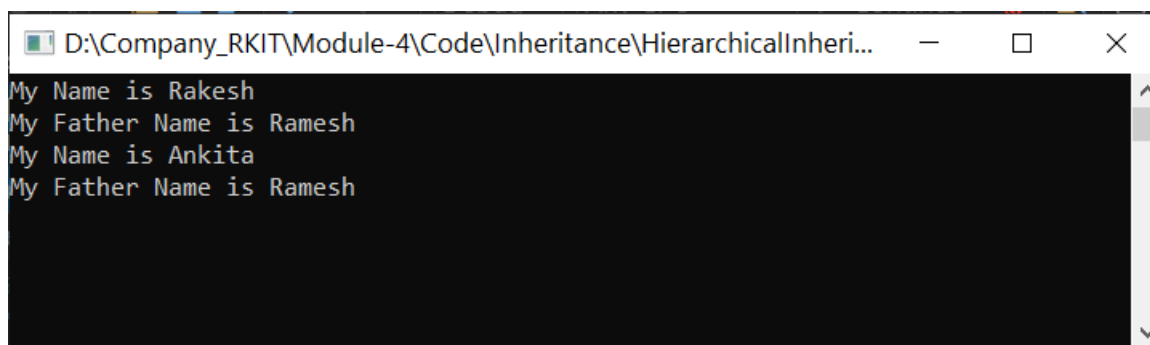
        Daughter objdaughter = new Daughter();

        // Displaying Daughter Name and Father Name for Daughter
        objdaughter.DaughterName();
        objdaughter.FatherName();

        Console.Read();
    }
}
}

```

Output:



```

D:\Company_RKIT\Module-4\Code\Inheritance\HierarchicalInheri...
My Name is Rakesh
My Father Name is Ramesh
My Name is Ankita
My Father Name is Ramesh

```

Multilevel Inheritance:

When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.

Code:

```
using System;
```

```

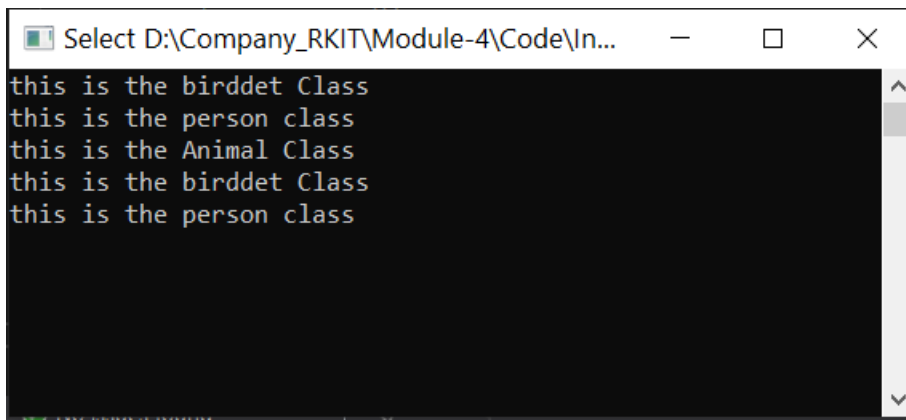
namespace MultilevelInheritance
{
    class Program
    {
        //base class
        public class Person
        {
            public string persondet()
            {
                return "this is the person class";
            }
        }
        //Derived class
        public class Bird : Person
        {
            public string birddet()
            {
                persondet();
                return "this is the birddet Class";
            }
        }
        //Derived class
        public class Animal : Bird
        {
            public string animaldet()
            {
                persondet();
                birddet();
                return "this is the Animal Class";
            }
        }
    }

    static void Main(string[] args)
    {
        Bird objbird = new Bird();
        // Calling a birddet() and persondet() for Bird
        String a = objbird.birddet();
        String b = objbird.persondet();
        Console.WriteLine(a);
        Console.WriteLine(b);

        Animal objanimal = new Animal();
        // Calling a animaldet() ,birddet() and persondet() for Animal
        String c = objanimal.animaldet();
        String d = objanimal.birddet();
        String e = objanimal.persondet();
        Console.WriteLine(c);
        Console.WriteLine(d);
        Console.WriteLine(e);
        Console.Read();
    }
}

```

Output:

A screenshot of a Windows file explorer window. The title bar shows the path "Select D:\Company_RKIT\Module-4\Code\In...". The main area displays a list of files: "this is the birddet Class", "this is the person class", "this is the Animal Class", "this is the birddet Class", and "this is the person class". The window has standard Windows controls (minimize, maximize, close) in the title bar.

Multiple Inheritance:

C# does not support multiple inheritances of classes. To overcome this problem we can use interfaces. Multiple inheritance means when a class is derived from multiple classes.

Code:

```
using System;

namespace MultipleInheritance
{
    public interface IA //ineterface 1
    {
        string setImgs(string a);
    }
    public interface IB //Interface 2
    {
        int getAmount(int Amt);
    }
    public class ICar : IA, IB //implementatin
    {
        public int getAmount(int Amt)
        {
            return 10;
        }
        public string setImgs(string a)
        {
            return "this is the car";
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            ICar objicar = new ICar();
            int a = objicar.getAmount(10);
            Console.WriteLine(a);
            String b = objicar.setImgs("ic object");
            Console.WriteLine(b);
            Console.Read();
        }
    }
}
```

Output:

D:\Company_RKIT\Module-4\Code\Inheritance\Multi...

10
this is the car